FORMAL REASONING IN SOFTWARE-DEFINED NETWORKS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Mark Reitblatt

January 2017

FORMAL REASONING IN SOFTWARE-DEFINED NETWORKS

Mark Reitblatt, Ph.D.

Cornell University 2017

This thesis presents an end-to-end approach for building computer networks that can be reasoned about and verified formally. In it, we present a high-level specification language for describing the desired forwarding behavior of networks based on regular expressions over network paths, as well as a tool that automatically verifies network forwarding policies; an approach to building formally verified compilers and runtimes for forwarding policies written in a network programming language that preserve the semantics of the source policy; and a technique for updating network configurations while preserving correctness.

BIOGRAPHICAL SKETCH

Mark Reitblatt was born, raised, and mostly educated in the great State of Texas. He attended the University of Texas (at Austin), receiving a Bachelors of Science in Pure Mathematics and a Bachelors of Science in Computer Science, with Honors. Then, after a short, but pleasant, stint at Intel, he joined the Computer Science PhD program at Cornell University.

This document is dedicated to all Cornell graduate students.

"Fight the power. We've got to fight the powers that be."

—Public Enemy

Please do not print this document unless absolutely necessary.

ACKNOWLEDGEMENTS

I would like to thank my advisor Nate for his infinite patience and support, Dexter Kozen for teaching me more about theory than I could possibly remember, and Bob Constable for always having an open door and endless enthusiasm for the questions and intellectual wonderings of a junior PhD student. My academic journey would have been deeply impoverished without the (sometimes cruel) tutelage of the inimitable Cornell Systems Lunch, which was (I hope) the site of my progression from truly terrible to acceptably adequate presenter. The crowd changed over the years, but a few faces stand out for their consistently incisive and inspiring insight and questioning: Robbert van Renesse, Fred Schneider[1], and Gün Sirer. I would also like to thank my collaborators for helping "pull me through the hole" on our papers: Marco Canini, Arjun Guha, Kostas Mamouras, Jen Rexford, Cole Schlesinger, Alexandra Silva, and David Walker.

My 5 years in Ithaca were made infinitely more enjoyable by the company and friendship of too many individuals to name here. Key among them was my constant roommate and friend, Eoin O'Mahony, and our merry changing band of roommates and drinking buddies, including but not limited to: Diarmuid Cahalane, Sam Hopkins, Jonathan DiLorenzo, Steffen Smolka, Rahmtin Rotabi, Daniel Freund, and Vasu Raman. Thanks also to my Ithaca Kickball team for giving me an *extra*-Cornell social outlet: Katie Moring, Lesley Middleton, Lynn Vincent, Jamie Schmohe, the Brothers Manning, Jess Gaby, Ian Dunham, Melinda Liptak, Danielle Morgan, and Matt Morgan. I am similarly grateful to my "Vet school friends" for an escape from the overly testosterone drench CS department: Marina Shepelev, Jane Park, Susan Smith, Jenna Goldhaber, Becca Vogel, and Jessica Chandler. And last, but certainly not least, my ever-present labmate, triathlon co-trainer, running mate, and eternal source of entertainment, Hussam Abu-Libdeh. Ithaca was never the same after you left buddy.

I would not be where I am today without the unimaginably rich intellectual climate of my

[1]Not to be confused with the unmustachioed lead singer of the B-52's

undergraduate studies. My Austin housemates (and pseudo-housemates): Gilbert Bernstein, Yonatan Bisk, Justin Hilburn, Tarun Nimmagadda, Jeremy Powell, Mickey Ristroph, and Gil Slade. And, of course, the inimitable Lorenzo Avisi. The technical aesthetic you inculcated has persisted to this day, and I hope that the work in this thesis can live up to it.

Finally, I would like to thank the Texas taxpayers and supporters of the University of Texas System. You've created one of the finest public education institutions anywhere in the world, and I would not be here without it.

Work supported in part by the National Science Foundation under grants CNS-1111698 *High-Level Language Support for Trustworthy Networks* and CNS-1413972 *Programmable Inter-Domain Observation and Control*, and the Office of Naval Research under grant N00014-12-1-0757 *Networks Opposing Botnets (NoBot) II*. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reect the views of the National Science Foundation or the Office of Naval Research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer networks are one of the great engineering contributions of the past century. If the 19th century was the era of the *Pax Brittanica* and global trade, then the 20th will almost certainly be remembered as the century of global communication, thanks in large part to the incredible global network we call the Internet. The Internet has grown at a breath-taking rate: at the time of the author's birth, in 1987, the Internet was little more than a novelty, used only by computer science academics, military research organizations, and handful of bleeding-edge enthusiasts. By the time of this thesis, 30 years later, the Internet is at the core of the modern economy. It is difficult to overstate the importance of computer networks to modern life: every Fortune 500 company relies upon the Internet for its daily operations, if not for its *raison d'etre*, and the U.N. has even recommended that "ensuring universal access to the Internet should be a priority for all States" [88].

As computer networks have grown in importance, so too have our demands upon them. Modern computer networks provide an array of functionality, far beyond the original goal of providing a communication channel between remote systems. A sample of features the author experienced in a single day while writing this thesis included:

- A captive portal that prevents coffeeshop users from accessing the Internet until they've accepted the terms of use
- "Wi-fi offload", which transparently moves mobile subscribers from cellular data to wi-fi networks
- Network traffic acceleration for high-latency satellite Internet connections[1]

[1]Satellite Internet is predominantly provided by satellites in geostationary orbit. Because of the large distance that radio signals have to travel to reach the satellite and return to Earth, and the speed-of-light,

- Traffic isolation to prevent users of the same local network from accessing each other's machines

With all of this functionality comes corresponding complexity. Our network devices have a dizzying array of configuration options, and understanding the behavior of a full network config file is not for the faint of heart. More than ever, we need tools that can automatically analyze configurations and determine whether or not the resulting network will have a desired correctness property.

Formally reasoning about network configurations requires building models that relate those configurations to the resulting network behaviors. However, the configuration "languages" that network vendors provide not only lack a formal semantics (specification of meaning), they often lack even a detailed informal description of the intended behavior. As a result, the only model of a configuration language that we can rely upon is the actual implementation, often a large, highly complex piece of closed source code numbering in the millions of lines.

As a result, existing verification tools for networks have followed one of three approaches. *Post hoc* verification tools such as VeriFlow [52] or NetPlumber [48] check properties of the current network state, avoiding modeling configurations altogether, but therefore cannot predict how the behavior of the system will change as the configuration changes (*e.g.* will a new configuration preserve connectivity?). Static configuration analyzers such as rcc [20] can detect generic configuration errors, but cannot establish correctness of a configuration with respect to a specification of correctness (*e.g.* does the configuration allow only traffic from trusted hosts to reach the server?).[2] Finally, verifiers such as Propane[8] identify specific

unaccelerated browsing can take more than two seconds to even begin loading a web page in the best of circumstances.

[2]One analogy would be the difference between an automated spelling and grammar checker in a word

subsets of well-defined functionality amenable to formal analysis (*e.g.* analyzing whether network routers will prefer to listen to the routes advertised by one neighbor over another).

This thesis takes a different strategy: rather than try to apply formal modeling after the fact to systems which were not designed for reasoning or correctness (and have repeatedly defied such attempts), we start from a clean-slate approach where network devices support only core network functionality (matching, modifying, and forwarding packets), and show how we can build networks that are designed from the ground-up to support formal reasoning and verification. Instead of being configured, these networks are programmed. This allows us to apply programming languages techniques for building and verifying systems. We start with a network programming language with well-defined formal semantics, and show how to use the semantics to build an automated tool that can prove whether or not a given program correctly implements a specification. We then build a formal model of software-defined networks, and show how to use this model to build formally verified compilers and run times that provably convert the original network program into equivalent network states. Finally, we show how to update the network policy in such a way that reasoning and verification performed on the old and new policies is preserved by the network while transitioning. Network updates are a common source of network errors and being able to reason about the behavior of a network under update is essential to a verified system.

Contributions Verification starts with a specification of correctness. In Chapter 3, we present an expressive, well-defined language for specifying the correct forwarding behavior of networks, precisely and formally describe what it means for a network program (a declarative specification of a specific forwarding configuration) to satisfy its specification, and show how to build a verifier that automatically proves the correctness (or incorrectness) of a program

processor, and a peer reviewer. The former is capable of telling you whether your paper has writing mistakes, but the latter can tell you whether or not your paper accomplishes what it set out to do.

with respect to a given specification.

Verifying a network program has only limited utility if a correct program is translated and implemented incorrectly by a compiler or runtime. Indeed, several works have found correctness bugs in compilers and network controllers that would counter-act the benefits of verification (see *e.g.* [14] [35]). In Chapter 4, we show how to build a formally verified compiler and runtime for network programs that provably preserves the correctness of the input program. We also build a formal model of OpenFlow [70], the most popular SDN protocol, and describe a generic proof technique that can be adapted to verify future implementations.

The techniques described in Chapters 3 and 4 only apply to a single network program, but real network forwarding policies change over time, in response to changes in the network topology, traffic loads, application demands, *etc.*. In Chapter 5 we show how to update a network from one forwarding policy to another in such a way that verification performed on the original and final configurations is preserved by the network in the transition.

Each of these developments is demonstrated by a real system that has been implemented and publicly released under an open-source license. Each system is also accompanied by a formal model or semantics that precisely and clearly models its behavior. In addition, all of the theorems and code described in Chapter 4, and most of the theorems in Chapter 5 have been formally verified in the Coq proof assistant.

CHAPTER 2

BACKGROUND

2.1 Historical context

Right now, in 2015, we are at the beginning of a large-scale revolution in the world of networking. For more than 30 years, there has been little fundamental change in the way that we design, build, and maintain networks. If you transported a network administrator from 1985 forward to today, they would recognize the fundamental principles of network design from their own day, even if the names, protocols, and other details have changed. At the same time, the kinds of applications and sheer scale of networks has changed in ways that we could have never dreamt of before the explosion of the internet and internet services.

This traditional networking style was based on a distributed architecture of expensive hardware boxes (called routers or switches, depending upon increasingly irrelevant protocol distinctions) that coordinated and configured themselves through a myriad of distributed protocols. For the purpose of this thesis, the important thing to understand is that every aspect of these networks, from the physical hardware to the software running on them, to the protocols they use to coordinate, was built and controlled wholly by network vendors, not the network owners.

Over the decades, there have been many proposals for new network architectures that solve the problems of traditional networking, and have the flexibility to adapt to new applications and demands. One of the best known proposals, Active Networks [97], turned packets into programs by embedding a full Turing-complete language into packet headers. Switches were transformed into interpreters, and control over forwarding and other network functions was delegated completely to end-hosts.

2.2 Software-defined Networking

More recently, Software-defined Networking (henceforth "SDN") has arisen as a serious challenger to the status quo. SDN decouples the packet-processing functions of the data plane from the control plane through a logically centralized-controller[1] that manages the network directly by configuring the packet-handling mechanisms in the underlying switches. The key elements of the design are that the switch configuration protocol is open and standardized, and the essential network management software runs on a commodity server programmed by the network owner. This allows the development of reusable implementations of generic network functions and makes the network almost completely customizable.

SDN has already seen adoption far beyond what Active Networks or any other competing proposal ever saw. Some of the largest technology companies in the world (Google, Microsoft, Facebook, VMware, and more) have either already adopted it, or are developing products around it.

OpenFlow In this thesis, we will use the OpenFlow SDN protocol to implement our network programs.

In an OpenFlow network, switches connect to a central controller, which then directly configures their forwarding behavior by programming a *flow table*. A flow table is a list of *match-action* rules, consisting of a *match* pattern that describes packet headers, and a list

[1]The controller is *centralized* is the sense that a single architectural component controls a function, as opposed to being *decentralized* in which many components control their own functions independently. Physically, the controller may be implemented as many co-ordinating distributed controllers.

of *action* rules that dictate how to process matching packet. An example is

Priority	Pattern	Action
2	$\{\mathbf{dlDst} = \mathit{H1}\}$	$\{\!\lvert 10 \rvert\!\}$
1	$\star$	$\{\!\lvert\rvert\!\}$

This flow table has two entries: the first one matches packets whose destination MAC address (**dlDst**) is *H1*, and forwards them out port 10. The second rule matches all packets and forwards them on no ports, which drops them. The rules have priorities that disambiguate overlapping rules. Because the first rule has a higher priority, it applies before the second one. Therefore, this flow table forwards packets destined for MAC address *H1* out port 10, and drops all other packets. In the rest of this thesis, we will omit priorities when possible. Rules will be listed in priority order, with high priority to low priority, top to bottom. Note that the action field of a rule is a multiset: if the same port is repeated, then a packet is duplicated and forwarded out the same port multiple times. In addition to forwarding, the action field can modify the value of packet header fields, or send the packet to the controller.

The controller programs flow tables by sending sequences of messages to install rules on the switch. To install the first rule in the above flow table, the controller would send:

$$\mathbf{Add}\ 2\ \{\mathbf{dlDst} = \texttt{H2}\}\ \{\!\lvert 10 \rvert\!\}$$

Network programming languages The interfaces exposed by SDN protocols, in particular OpenFlow, are quite low-level, the networking equivalent of assembly. Recent work has proposed a number of high-level languages and language abstractions to simplify the task of developing correct, reliable SDN systems. See [15] and [23] for a survey of current developments in network programming languages.

In this thesis, we focus upon two programming languages in the Frenetic family: NetKAT

[4] and its predecessor NetCore [74]. NetKAT is introduced in this chapter, and NetCore is described before it is used in Chapter 4.

2.3 The NetKAT Programming Language

The network programming language NetKAT was developed by Anderson *et al.* [4]. NetKAT is an extension of Kleene algebra with tests (KAT), an algebraic system for program verification that combines Kleene Algebra (KA) with boolean algebra [57]. NetKAT offers a collection of intuitive constructs including: predicates over packets; primitives for modifying packet headers and encoding topologies; iterations; and sequential and parallel composition operators. The semantics is given in terms of a denotational model based on functions from packet histories to sets of packet histories (where a history records a packet's path through the network). In addition to the denotational semantics, NetKAT has a sound and complete equational deductive system.

2.3.1 Syntax

NetKAT [4] is based upon Kleene algebra with tests (KAT) [57], a generic equational system for reasoning about partial correctness of programs.

Kleene Algebra (KA) & Kleene Algebra with Tests (KAT) A *Kleene algebra* (KA) is an algebraic structure,

$$(K,\ +,\ \cdot,\ \star,\ 0,\ 1)$$

where K is an idempotent semiring under $(+, \cdot, 0, 1)$, and $p^* \cdot q$ is the least solution of the inequality $p \cdot r + q \leq r$, where $p \leq q$ is shorthand for $p + q = q$, and similarly for $q \cdot p^*$. A

Naturals	n	$\in$	$0 \mid 1 \mid 2 \mid \ldots$
Fields	x	$::=$	$x_1 \mid \cdots \mid x_k$
Packets	pk	$::=$	$\{f_1 = n_1, \cdots, f_k = n_k\}$
Histories	σ	$::=$	$\langle pk \rangle \mid pk{:}\sigma$

Tests	a	$::= \mathsf{1}$	*True*
		$\mid \mathsf{0}$	*False*
		$\mid x = n$	*Header test*
		$\mid a_1 + a_2$	*Disjunction*
		$\mid a_1 \cdot a_2$	*Conjunction*
		$\mid \neg a$	*Negation*
Actions	p	$::= a$	*Test*
		$\mid x \leftarrow n$	*Modification*
		$\mid p_1 + p_2$	*Parallel Composition*
		$\mid p_1 \cdot p_2$	*Sequential Composition*
		$\mid p^*$	*Iteration*
		$\mid \mathsf{dup}$	*Duplication*

Figure 2.1: NetKAT Syntax.

Kleene algebra with tests (KAT) is an algebraic structure,

$$(K,\ B,\ +,\ \cdot,\ \star,\ 0,\ 1,\ \neg)$$

where $\neg$ is a unary operator defined only on B, such that

- $(K,\ +,\ \cdot,\ \star,\ 0,\ 1)$ is a Kleene algebra,
- $(B,\ +,\ \cdot,\ \neg\ ,\ 0,\ 1)$ is a Boolean algebra, and
- $(B,\ +,\ \cdot,\ 0,\ 1)$ is a subalgebra of $(K,\ +,\ \cdot,\ 0,\ 1)$.

The elements of B and K are called *tests* and *actions*.

The axioms of KA and KAT (both elided here) capture natural conditions such as associativity of $\cdot$; see the original paper by Kozen for a complete listing [57].

NetKAT NetKAT [4] extends KAT with network-specific primitives for filtering, modifying, and forwarding packets, along with additional axioms for reasoning about programs built using those primitives. Formally, NetKAT is KAT with atomic actions and tests

$$x \leftarrow n \qquad\qquad x = n \qquad\qquad \mathsf{dup}$$

with the following meanings: The test $x = n$ tests whether field x of the current packet contains the value n; the assignment $x \leftarrow n$ assigns the value n to the field x in the current packet; and the action dup duplicates the last packet in the packet history, which keeps track of the path the packet takes through the network.

For example, the NetKAT expression

$$pt = 5 \cdot sw = 3 \cdot dst \leftarrow 192.168.1.5 \cdot pt \leftarrow 5$$

encodes the command: "For all packets located at port 5 of switch 3, set the destination address to 192.168.1.5 and forward it out on port 5."

2.3.2 Semantics

The standard semantics of NetKAT interprets expressions as packet-processing functions. As defined in Figure 2.1, a packet π is a record whose fields assign constant values n to fields x and a packet history is a nonempty sequence of packets $\pi_1 : \pi_2 : \cdots : \pi_k$, listed in order of youngest to oldest. Operationally, only the head packet π_1 exists in the network, but we keep track of the packet's history in the semantics to enable precise reasoning about behavior involving forwarding along different paths.

Formally, a NetKAT term p denotes a function

$$[\![p]\!] : \mathcal{H} \to 2^{\mathcal{H}},$$

where $\mathcal{H}$ is the set of all packet histories. Intuitively, the function $[\![p]\!]$ takes an input packet history σ and produces a set of output packet histories $[\![p]\!](\sigma)$, representing all of the packets that result from the forwarding function, and their associated histories.

The semantics of the primitive actions and tests in NetKAT are as follows. For a packet history $\pi{:}\sigma$ with head packet π,

$$
\begin{aligned}
[\![x \leftarrow n]\!](\pi{:}\sigma) &= \{\pi[n/x]{:}\sigma\} \\
[\![x = n]\!](\pi{:}\sigma) &= \begin{cases} \{\pi{:}\sigma\}, & \pi(x) = n \\ \emptyset, & \pi(x) \neq n \end{cases} \\
[\![\mathsf{dup}]\!](\pi{:}\sigma) &= \{\pi{:}\pi{:}\sigma\} \\
[\![\mathsf{1}]\!](\sigma) &= \{\sigma\} \\
[\![\mathsf{0}]\!](\sigma) &= \emptyset.
\end{aligned}
$$

where $\pi[n/x]$ denotes the packet π with the field x rebound to the value n. A test $x = n$ filters out (drops) the packet if the test is not satisfied and passes it through if it is. The dup construct duplicates the head packet π, yielding a fresh copy that can be modified by other constructs. Hence, in this standard model, the dup construct can be used to encode paths through the network, with each occurrence of dup marking an intermediate hop.

The operations ($+$, $\cdot$, *, $\neg$) are interpreted as follows:

$$
\begin{aligned}
[\![p + q]\!](\sigma) &= [\![p]\!](\sigma) \cup [\![q]\!](\sigma) \\
[\![p \cdot q]\!](\sigma) &= \bigcup_{\tau \in [\![p]\!](\sigma)} [\![q]\!](\tau) \\
[\![p^*]\!](\sigma) &= \bigcup_n [\![p^n]\!](\sigma)
\end{aligned}
$$

$$[\![\neg a]\!](\sigma) = \begin{cases} \{\sigma\}, & \text{if } [\![a]\!](\sigma) = \emptyset \\ \emptyset, & \text{if } [\![a]\!](\sigma) = \{\sigma\} \end{cases}$$

Note that $+$ behaves like disjunction when applied to tests and like union when applied to actions. Similarly, $\cdot$ behaves like conjunction when applied to tests and like sequential composition when applied to actions. Negation is only ever applied to tests, as is enforced by the syntax of the language.

2.3.3 Equational Theory

NetKAT has a sound and complete equational theory, based upon the equational theory of KAT. This means that two NetKAT terms are semantically equivalent (denote the same function) iff there is a proof of equivalence using the NetKAT axioms.

The NetKAT axioms, shown in Fig. 2.2, consist of the axioms for Kleene Algebra (starting with KA-*), the axioms for a Boolean Algebra (starting with BA-*), and NetKAT-specific Packet Algebra axioms (starting with PA-*) describing the interaction between packet modifications and packet tests.

The Packet Algebra axioms say that modifications or filters on disparate fields commute (PA-MOD-MOD-COMM and PA-MOD-FILTER-COMM); filters commute with dup (PA-DUP-FILTER-COMM); modifying a packet field to a specific value and then testing for that same value is the same as only performing the modification, and vice versa (PA-MOD-FILTER and PA-FILTER-MOD); when modifying the same field twice, the second modification "wins" (PA-MOD-MOD); testing the same field for different values is always false (PA-CONTRA); and that the disjunction of all possible tests for a single field is always satisfied (PA-MATCH-ALL).

2.3.4 Language Model

We stated that the above axioms are sound and complete for NetKAT. Soundness is easy enough to show using the semantics, but proving completeness requires different tools. Following the standard approach, Anderson *et al.* [4] develop a *language model*, a semantics in which terms are interpreted as sets of "strings" (formally, elements in a monoid). For NetKAT, these are "reduced" strings of the form

$$\alpha p_0 \mathsf{dup} \pi_1 \mathsf{dup} \pi_2 \cdots \pi_{n-1} \mathsf{dup} \pi_n, \quad n \geq 0,$$

where α is a *complete test* $x_1 = n_1 \cdot \ldots \cdot x_k = n_k$, π_i is a *complete assignment* $x_1 \leftarrow n_1 \cdot \ldots \cdot x_k \leftarrow n_k$, and each of the fields is x_k for exactly one k. We will write At for the set of complete tests, and P for the set of complete assignments. The set of reduced strings is $\mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$, where dup is the singleton set containing dup; $A \cdot B$ denotes string concatenation (lifted to sets of strings)[2]; and A^* denotes $\cup_i A^i$, where $A^{i+1} \triangleq A \cdot A^i$ and $A^0 \triangleq \{\epsilon\}$ for ϵ the empty string.

Every NetKAT expression is equivalent to a *reduced expression* in which every test is a complete test and every assignment is a complete assignment. The complete tests are the atoms (minimal nonzero elements) of the Boolean algebra generated by the primitive tests. Complete tests and complete assignments are in one-to-one correspondence.

The full language model for NetKAT is defined over the reduced expressions and is shown in Fig. 2.3. The language denoted by a complete test α is the singleton set containing the reduced string $\alpha \cdot \pi_\alpha$, where π_α is the complete assignment corresponding to α. The language denoted by a complete assignment is the set of reduced string $\alpha \cdot \pi$ for every complete test

[2]Note: string concatenation is different from the guarded concatenation used in the language model. See Fig. 2.3 for guarded concatenation

α. The language of $p + q$ is the union of the languages of p and q; the language of a sequential composition is the guarded concatenation of the languages; and the language of p^* is the union of all finite iterates of the language of p^n. The semantics of dup requires some explanation: dup is supposed to make a head copy of the head packet. Thus, it takes the head packet α, and copies it across the place-holder dup as π_α.

See the original paper on NetKAT [4] for a comprehensive treatment of the language model, including proofs of the claims above.

Kleene Algebra Axioms

$p + (q + r)$	$\equiv$	$(p + q) + r$	KA-PLUS-ASSOC
$p + q$	$\equiv$	$q + p$	KA-PLUS-COMM
$p + \mathsf{0}$	$\equiv$	p	KA-PLUS-ZERO
$p + p$	$\equiv$	p	KA-PLUS-IDEM
$p \cdot (q \cdot r)$	$\equiv$	$(p \cdot q) \cdot r$	KA-SEQ-ASSOC
$\mathsf{1} \cdot p$	$\equiv$	p	KA-ONE-SEQ
$p \cdot \mathsf{1}$	$\equiv$	p	KA-SEQ-ONE
$p \cdot (q + r)$	$\equiv$	$p \cdot q + p \cdot r$	KA-SEQ-DIST-L
$(p + q) \cdot r$	$\equiv$	$p \cdot r + q \cdot r$	KA-SEQ-DIST-R
$\mathsf{0} \cdot p$	$\equiv$	$\mathsf{0}$	KA-ZERO-SEQ
$p \cdot \mathsf{0}$	$\equiv$	$\mathsf{0}$	KA-SEQ-ZERO
$\mathsf{1} + p \cdot p^*$	$\equiv$	p^*	KA-UNROLL-L
$q + p \cdot r \leq r$	$\Longrightarrow$	$p^* \cdot q \leq r$	KA-LFP-L
$\mathsf{1} + p^* \cdot p$	$\equiv$	p^*	KA-UNROLL-R
$p + q \cdot r \leq q$	$\Longrightarrow$	$p \cdot r^* \leq q$	KA-LFP-R

Additional Boolean Algebra Axioms

$a + (b \cdot c) \equiv (a + b) \cdot (a + c)$		BA-PLUS-DIST
$a + \mathsf{1} \equiv \mathsf{1}$		BA-PLUS-ONE
$a + \neg a \equiv \mathsf{1}$		BA-EXCL-MID
$a \cdot b \equiv b \cdot a$		BA-SEQ-COMM
$a \cdot \neg a \equiv \mathsf{0}$		BA-CONTRA
$a \cdot a \equiv a$		BA-SEQ-IDEM

Packet Algebra Axioms

$f \leftarrow n \cdot f' \leftarrow n' \equiv f' \leftarrow n' \cdot f \leftarrow n, \text{ if } f \neq f'$	PA-MOD-MOD-COMM
$f \leftarrow n \cdot f' = n' \equiv f' = n' \cdot f \leftarrow n, \text{ if } f \neq f'$	PA-MOD-FILTER-COMM
$\mathsf{dup} \cdot f = n \equiv f = n \cdot \mathsf{dup}$	PA-DUP-FILTER-COMM
$f \leftarrow n \cdot f = n \equiv f \leftarrow n$	PA-MOD-FILTER
$f = n \cdot f \leftarrow n \equiv f = n$	PA-FILTER-MOD
$f \leftarrow n \cdot f \leftarrow n' \equiv f \leftarrow n'$	PA-MOD-MOD
$f = n \cdot f = n' \equiv \mathsf{0}, \text{ if } n \neq n'$	PA-CONTRA
$\sum_i f = i \equiv \mathsf{1}$	PA-MATCH-ALL

Figure 2.2: NetKAT axioms

Language model: $G(p) \subseteq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$

$$
\begin{aligned}
G(\alpha) &= \{\alpha \cdot \pi_\alpha\} \\
G(\pi) &= \{\alpha \cdot \pi \mid \alpha \in \mathsf{At}\} \\
G(p+q) &= G(p) \cup G(q) \\
G(p \cdot q) &= G(p) \diamond G(q) \\
G(\mathsf{dup}) &= \{\alpha \cdot \pi_\alpha \cdot \mathsf{dup} \cdot \pi_\alpha \mid \alpha \in \mathsf{At}\} \\
G(p^*) &= \textstyle\bigcup_{n \geq 0} G(p^n)
\end{aligned}
$$

Guarded concatenation

$$
\alpha \cdot p \cdot \pi \diamond \beta \cdot q \cdot \pi' = \begin{cases} \alpha \cdot p \cdot q \cdot \pi' & \text{if } \beta = \alpha_\pi \\ \text{undefined} & \text{if } \beta \neq \alpha_\pi \end{cases}
$$

$$
A \diamond B = \{p \cdot q \mid p \in A,\ q \in B\}
$$

Figure 2.3: NetKAT language model

CHAPTER 3

VERIFYING NETWORK PROGRAMS

"Seek simplicity and distrust it."

—Alfred North Whitehead

In this chapter, we introduce a novel specification language for network forwarding properties (Pathetic), and show how to build a formal verification tool that automatically analyzes NetKAT policies for correctness with respect to a Pathetic specification.

3.1 Introduction

Network configurations have long been a source of serious bugs and misbehavior in real-world networks. Written in arcane, poorly specified (often unspecified) languages, they defied meaningful verification. Before the development of a static configuration analyzer (the router configuration checker *rcc*) for BGP routers, the status quo in practice was run-time testing on operational networks [20]. Even *rcc* was only capable of detecting generic faults in configurations (*e.g.* learning unusable paths) and could not prove functional correctness. More recent verification tools (*e.g.* AntEater [64], HSA [49], or VeriFlow [52]) have focused on stronger correctness properties, but are based on *ad-hoc* foundations, and lack well-defined specification languages. For example, the NetPlumber [48] verifier includes a specification language for describing network paths, but the language itself does not have a semantics and lacks a precise description of what it means for a configuration to satisfy a specification.

This is not just a pedantic, academic point: without a precise semantics that describes the meaning of a specification and what it means to satisfy it, users cannot reason about

the specification itself, nor can they have confidence in any purported verification performed against it. Moreover, anyone who builds a verifier based on such a language has no way of knowing whether they have even implemented it correctly.

In this chapter, we take a different approach. We present a specification language (Pathetic) with clear, precise formal semantics, and show how to use it to encode the requirements of an example network program. We then formally describe what it means for a network implementation (in the form of a NetKAT program) to satisfy a Pathetic specification, and show how to design and build a verifier that automatically decides satisfaction.

More specifically, in this chapter, we:

- Define the syntax and semantics of the Pathetic specification language
- Extend NetKAT with new operators (intersection and complement) to enable translation from Pathetic (NetKAT$(-, \cap)$)
- Define a semantics-preserving translation from Pathetic to NetKAT$(-, \cap)$
- Extend the equational and automata theories of NetKAT to NetKAT$(-, \cap)$
- Use the extended theories of NetKAT$(-, \cap)$ to build a decision procedure for Pathetic satisfaction

An early version of the Pathetic language appeared in [86].

3.2 Example

We will use the following running example through out the chapter to introduce Pathetic and demonstrate its features.

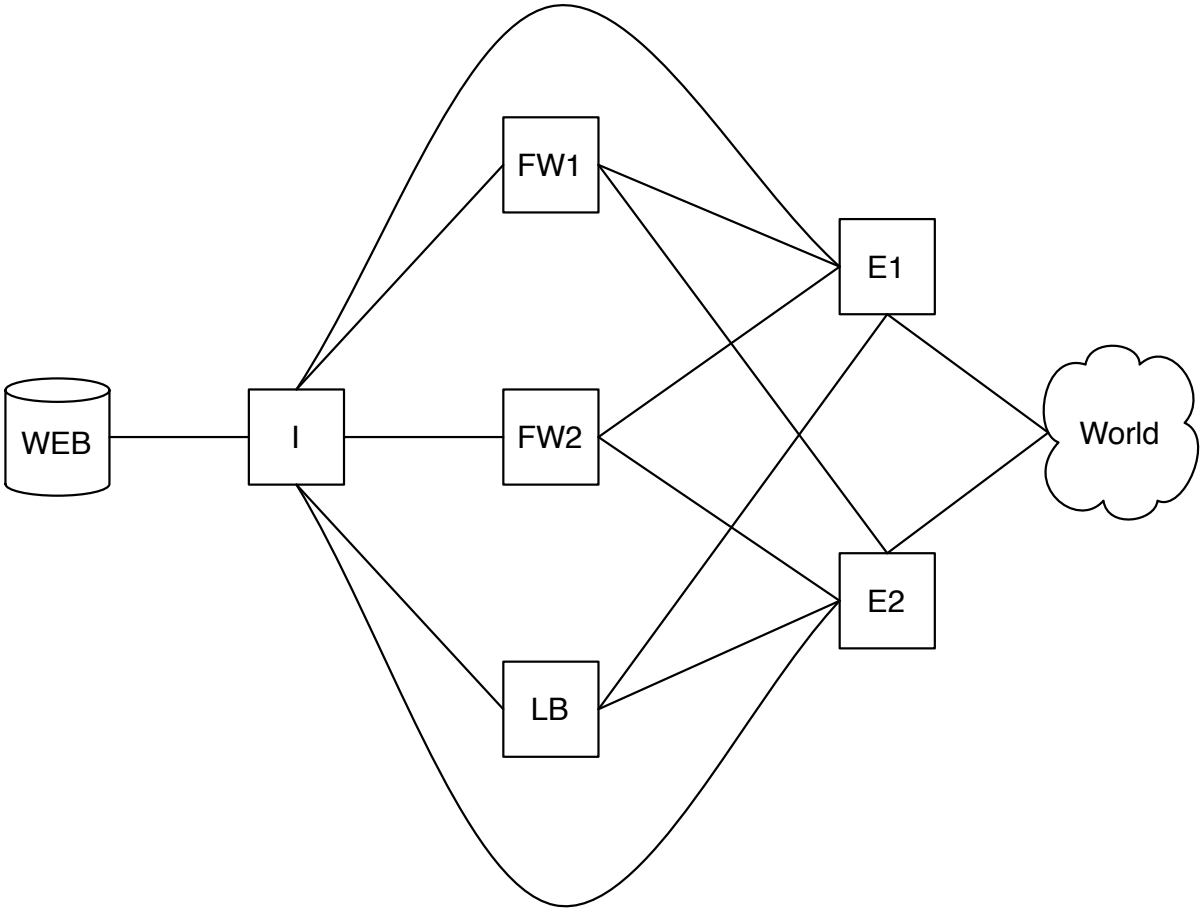


Figure 3.1: Example topology.

Consider the network shown in Figure 3.1. It consists of one attached network, World, a webserver Web, an ingress switch I, two firewalls FW1 and FW2, a load balancer LB, and two egress switches E1 and E2. This network is intended to provide connectivity between Web and World, subject to a security policy and a load balancing policy. The security policy, which we will denote by ϕ_{FW}, states that all outbound traffic (traffic originating at Web and destined for World) must traverse a firewall before reaching an egress switch. The load balancing policy, denoted by ϕ_{LB}, states that all inbound traffic destined for the webserver (traffic originating in World and destined for an address denoted Web), must traverse the load balancer LB before reaching the ingress switch I.

The network is implemented with SDN switches, managed by a single controller Controller as shown in Figure 3.2[1]. For simplicity, we assume the switches are connected to the controller via a separate control network, independent from the network in the example. Because this chapter deals only with static network configurations and is agnostic to the mechanism used to implements them in the network, the techniques in this chapter are equally applicable when the switches are connected to the controller via the primary data network (so called "inband control").

The network is managed by an application, which receives network events (switches connecting, responses to queries, etc) and sends out network programs written in the network programming language NetKAT. A compiler takes the network programs, converts them into switch rules, and gives them to the controller to implement in the network.

To implement the load-balancing policy, the network application might initially install this NetKAT policy:

$$p_{\mathsf{LB}} \triangleq \quad tpDst = 80 \cdot nwDst = \mathsf{WEB} \cdot \begin{pmatrix} (sw = \mathsf{E1} + sw = \mathsf{E2}) \cdot pt \leftarrow \mathsf{LB} \\ +sw = \mathsf{LB} \cdot pt \leftarrow \mathsf{I} \\ +sw = \mathsf{I} \cdot pt \leftarrow \mathsf{WEB} \end{pmatrix}$$

Informally, this policy should be read as "p_{LB} is defined to be equal to the policy that matches packets with a destination port of 80 and a destination address of WEB, and if the current switch is E1 or E2 then sends the packet to LB, and if the current switch is LB then sends the packet to I, and if the current switch is I then sends the packet to WEB".

We briefly review the syntax and semantics of NetKAT here, but for full details, see

[1]For review of what an SDN network is, see Section 2.2

Chapter 2 or Anderson *et al.* [4]. The policy consists of several terms, composed with the sequential composition operator $\cdot$ and the parallel composition operator $+$. The first term is a predicate (or filter), $tpDst = 80$, that tests the destination port ($tpDst$) of packets, letting them pass through if it is equal to 80, or dropping them if not. This test is sequentially composed with another predicate, $nwDst = \mathsf{WEB}$, that tests the destination IP address for equality with the web server WEB's IP address. Notice that sequentially composing predicates is equivalent to taking their conjunction. Next, we compose these two predicates with the parallel composition of several terms. The first term in the parallel composition (or union) matches packets that are at either $\mathsf{E1}$ or $\mathsf{E2}$ (parallel composition of predicates is the same as their disjunction), and sends them to the port connected to LB. Similarly, the next terms matches packets on LB and sends them to the port connected to I. Finally, the last term matches packets that have arrived at I and sends them to the port connected to WEB.

This policy describes the switch forwarding behavior, but NetKAT programs actually encode the full network behavior, including the topology. Generally, the topology term is provided by the controller and composed with the switch forward term. A snippet of the topology term for our network would be written as follows:

$$
\begin{aligned}
t \triangleq\ & sw = \mathsf{E1} \cdot pt = \mathsf{FW1} \cdot sw \leftarrow \mathsf{FW1} \cdot pt \leftarrow \mathsf{E1} \\
& + sw = \mathsf{E1} \cdot pt = \mathsf{FW2} \cdot sw \leftarrow \mathsf{FW2} \cdot pt \leftarrow \mathsf{E1} \\
& + sw = \mathsf{E1} \cdot pt = \mathsf{LB} \cdot sw \leftarrow \mathsf{LB} \cdot pt \leftarrow \mathsf{E1} \\
& + sw = \mathsf{E1} \cdot pt = \mathsf{I} \cdot sw \leftarrow \mathsf{I} \cdot pt \leftarrow \mathsf{E1} \\
& \ldots
\end{aligned}
$$

To construct the term describing the full network, we would compose the switch forwarding term, the term dup, and the topology term, and iterate them using the * operator:

$$(p_{\mathsf{LB}} \cdot \mathsf{dup} \cdot t)^*$$

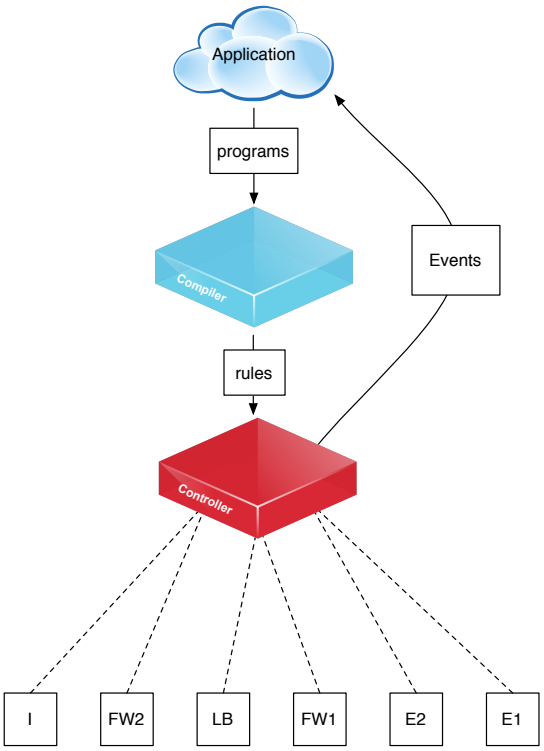


Figure 3.2: Running example control architecture.

The constant dup and requires explanation. NetKAT's semantics is given as a function from *histories* (non-empty lists of packet headers) to sets of *histories*. Atomic NetKAT terms (such as header predicates and modifications) work on the head packet of the current history. The operation dup duplicates the head packet, putting the new copy on the top of the list. This is how histories remember past values of packets, and how the semantics models paths through the network. In practice, programmers do not program with dup directly; it is instead inserted in the appropriate place along with the topology term.

3.3 The Pathetic specification language

In this section we introduce Pathetic, a specification language based on regular expressions over network paths with wildcards, and demonstrate its use through the running example introduced in the previous section. While Pathetic is, in one sense, strictly *less* expressive than NetKAT[2], it is in fact specialized to serve a different role. In Pathetic, alternation is interpreted *disjunctively* (*i.e.* choose one of the following), while in NetKAT alternation is interpreted *conjunctively* (*i.e.* perform *all* of the following). Thus, a Pathetic program can specify multiple possible NetKAT implementations. As the examples in this chapter will show, by utilizing the wildcard operator and the iteration operator, a Pathetic program can succinctly describe only the essential details of a path that matter for correctness.

Load balancing policy Consider the load balancer requirement from the running example: all web traffic destined for the server WEB must traverse LB before reaching the ingress switch I. We can write down this specification in Pathetic as ϕ_{LB}:

$$\phi_{\mathsf{LB}} \triangleq \phi_{\mathsf{WEB}} \uplus \phi_{\neg\mathsf{WEB}}$$

where

$$\phi_{\mathsf{WEB}} \triangleq (tpDst = 80 \cdot nwDst = \mathsf{WEB}) \Rightarrow (\neg\mathsf{WEB})^*.\mathsf{LB}.(\star^*).\mathsf{WEB}$$

$$\phi_{\neg\mathsf{WEB}} \triangleq \neg(tpDst = 80 \cdot nwDst = \mathsf{WEB}) \Rightarrow \star^*$$

The policy ϕ_{LB} is written as the union of two *atomic* Pathetic policies, ϕ_{WEB} and ϕ_{other}. Atomic policies such as ϕ_{WEB} consist of a predicate ($tpDst = 80 \cdot nwDst = \mathsf{WEB}$) describing the packets that the policy applies to, and a path expression ($(\neg\mathsf{WEB})^*.\mathsf{LB}.(\star^*).\mathsf{WEB}$) describing

[2]All Pathetic programs are translatable into semantically NetKAT programs, but not vice versa

the valid paths. Predicates are expressed using the syntax of the NetKAT predicate language. $(tpDst = 80 \cdot nwDst = \mathsf{WEB})$ matches web traffic $(tpDst = 80)$ that is destined for the web server $(nwDst = \mathsf{WEB})$. The path expression consists of four parts, joined together with the sequence operator ".". In the first part, we use the shorthand $\neg\mathsf{WEB}$ (formally equal to $\star \cap \overline{\mathsf{WEB}}$ where $\star$ is a wildcard denoting any path of length one), which matches all paths of length one other than [WEB]. The policy P^* denotes iteration zero or more times of the path expression P, thus $(\neg\mathsf{WEB})^*$ matches paths of any length that do not traverse WEB. After this initial path, the packet must traverse LB and then is allowed to take any path $(\star^*)$ that ends at WEB.

Finally, because Pathetic policies denote total specifications of network behavior (unmatched packets get dropped), we union this with a fall-through policy $\phi_{\neg\mathsf{WEB}}$ that allows non-web traffic to take any path through the network.

Firewall policy Similarly, we can write down the policy requiring that all outbound traffic traverse either FW1 or FW2 before leaving the network by decomposing it into three parts. First, all packets (1 denotes the predicate "true") are allowed to take any path that ends inside the network $(\neg(\mathsf{E1}|\mathsf{E2}))$:

$$\phi_{internal} \triangleq 1 \Rightarrow (\star^*).\neg(\mathsf{E}_1|\mathsf{E}_2)$$

Second, all traffic starting inside the network is allowed to take a path that traverses FW1 or FW2 before leaving the network (via one of the egress switches):

$$\phi_{internal-external} \triangleq 1 \Rightarrow (\neg(\mathsf{E}_1|\mathsf{E}_2))^*.(\mathsf{FW1}|\mathsf{FW2}).(\star^*).(\mathsf{E}_1|\mathsf{E}_2)$$

Third, we allow traffic between the egress switches to take arbitrary paths between them:

$$\phi_{external-external} \triangleq 1 \Rightarrow (\mathsf{E}_1|\mathsf{E}_2).(\star^*).(\mathsf{E}_1|\mathsf{E}_2)$$

Finally, we combine these policies with the union operator to allow any of the paths to be used:

$$\phi_{\mathsf{FW}} \triangleq \phi_{internal} \uplus \phi_{internal-external} \uplus \phi_{external-external}$$

Combined example To enforce both the firewall policy and the previous load balancing policy, we combine them using the intersection operator, to impose the requirements of both policies:

$$\phi_{\mathsf{FW-LB}} \triangleq \phi_{\mathsf{FW}} \pitchfork \phi_{\mathsf{LB}}$$

The resulting policy $\phi_{\mathsf{FW-LB}}$ enforces that all outbound traffic traverses the firewall, and that all inbound web traffic traverses the load balancer before arriving at the webserver.

3.3.1 Pathetic syntax and semantics

The syntax of Pathetic programs is shown in Figure 3.3a. Atomic Pathetic programs ($a \Rightarrow P$) consist of two parts: a regular expression over network elements describing a set of valid paths (P), and a predicate defining the set of packets that the regular expression applies to (a). Atomic path regular expressions are either the empty path (ϵ), the empty set of paths ($\emptyset$), a constant path of length one (S for some switch S) or a wildcard path of length 1 ($\star$). Regular expressions can be combined using sequential composition ($P.P'$), non-deterministic choice ($P|P'$), complement ($\overline{P}$), conjunction ($P \cap P'$), or iteration (P^*). Compound Pathetic programs are constructed with the union of two programs ($\phi \uplus \phi'$), which denotes the disjunction of the restrictions of ϕ and ϕ', or intersection ($\phi \pitchfork \phi'$), which denotes the conjunction of the restrictions of ϕ and ϕ'.

Syntax

Path exp.	P	$::= \epsilon$	*Empty path*
		$\mid \emptyset$	*Empty set*
		$\mid S$	*Explicit switch S*
		$\mid \star$	*Wildcard hop*
		$\mid \overline{P}$	*Complement*
		$\mid P.P$	*Sequencing*
		$\mid P \mid P$	*Alternation*
		$\mid P \cap P$	*Intersection*
		$\mid P^*$	*Iteration*
Fields	f	$::= f_1 \mid \cdots \mid f_k$	
Predicates	a, b	$::= \mathsf{1}$	*Identity*
		$\mid \mathsf{0}$	*Drop*
		$\mid f = n$	*Test*
		$\mid a + b$	*Disjunction*
		$\mid a \cdot b$	*Conjunction*
		$\mid \neg a$	*Negation*
Program	ϕ	$::= a \Rightarrow P$	*atomic policy*
		$\mid \phi_1 \uplus \phi_2$	*policy union*
		$\mid \phi_1 \pitchfork \phi_2$	*policy intersection*

(a)

Path semantics

$$
\begin{aligned}
[\![P]\!] &\in 2^{\mathtt{Sw}^*} \\
[\![\epsilon]\!] &\triangleq \{[]\} \\
[\![\emptyset]\!] &\triangleq \{\} \\
[\![S]\!] &\triangleq \{[S]\} \\
[\![\star]\!] &\triangleq \{[S] \mid S \in \mathtt{Sw}\} \\
[\![\overline{P}]\!] &\triangleq 2^{\mathtt{Sw}^*} \setminus [\![P]\!] \\
[\![P.P']\!] &\triangleq [\![P]\!] \diamond [\![P']\!] \\
[\![P \mid P']\!] &\triangleq [\![P]\!] \cup [\![P']\!] \\
[\![P \cap P']\!] &\triangleq [\![P]\!] \cap [\![P']\!] \\
[\![P^*]\!] &\triangleq \textstyle\bigcup_{n \geq 0} F^n \\
&\text{where } F^0 = \{[]\} \\
&\text{and } F^{i+1} = [\![P]\!] \diamond F^i \\
&\text{and } A \diamond B = \{ab \mid a \in A \wedge b \in B\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\phi]\!] &\in \mathcal{PK} \to 2^{\mathtt{Sw}^*} \\
[\![a \Rightarrow P]\!]\ pk &\triangleq \begin{cases} [\![P]\!] & \text{if } [\![a]\!]\ pk \\ \{\} & \text{o.w.} \end{cases} \\
[\![\phi_1 \uplus \phi_2]\!]\ pk &\triangleq [\![\phi_1]\!]\ pk \cup [\![\phi_2]\!]\ pk \\
[\![\phi_1 \pitchfork \phi_2]\!]\ pk &\triangleq [\![\phi_1]\!]\ pk \cap [\![\phi_2]\!]\ pk
\end{aligned}
$$

(b)

Figure 3.3: Pathetic syntax and semantics.

Pathetic semantics The semantics of Pathetic is shown in Figure 3.3b. Pathetic programs denote functions from packets to sets of allowable paths[3]. $\phi \uplus \phi'$ denotes the point-wise union of ϕ and ϕ', and $\phi \pitchfork \phi'$ denotes the point-wise intersection.

Let's look at ϕ_{LB} from our running example, and see how it behaves on a web packet destined for WEB:

$$[\![\phi_{\mathsf{LB}}]\!]\ pk = [\![\phi_{\mathsf{WEB}}]\!]\ pk \cup [\![\phi_{\neg\mathsf{WEB}}]\!]\ pk$$

[3]Notice that this rules out the possibility of packet modifications

$$= \begin{cases} [\![(\neg \mathsf{WEB})^*.\mathsf{LB}.(\star^*).\mathsf{WEB}]\!] & \text{if } [\![(tpDst = 80 \cdot nwDst = \mathsf{WEB})]\!]\ pk \\ [\![\star^*]\!] & \text{o.w.} \end{cases}$$

$$= [\![(\neg \mathsf{WEB})^*.\mathsf{LB}.(\star^*).\mathsf{WEB}]\!]$$

$$= [\![(\neg \mathsf{WEB})^*]\!] \diamond [\![\mathsf{LB}]\!] \diamond [\![(\star^*)]\!] \diamond [\![\mathsf{WEB}]\!]$$

$$= [\![(\star \cap \overline{\mathsf{WEB}})^*]\!] \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\}$$

$$= \cup_i [\![\star \cap \overline{\mathsf{WEB}}]\!]^i \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\}$$

$$= \cup_i \left([\![\star]\!] \cap [\![\overline{\mathsf{WEB}}]\!]\right)^i \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\}$$

$$= \cup_i \left(\mathtt{Sw} \cap (\mathtt{Sw}^* \setminus \{[\mathsf{WEB}]\})\right)^i \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\}$$

$$= \cup_i \left(\mathtt{Sw} \setminus \{[\mathsf{WEB}]\}\right)^i \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\}$$

$$= \{P \mid \mathsf{WEB} \notin P\} \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\}$$

$$= \{P \cdot [\mathsf{LB}] \cdot P' \cdot [\mathsf{WEB}] \mid \mathsf{WEB} \notin P\}$$

So, this gives us the set of paths that take any route to LB not through WEB, and then take any route to WEB, which is exactly what we wanted.

3.3.2 Relating Pathetic to NetKAT

This section assumes familiarity with the NetKAT semantics language model. For more details, see Chapter 2.

Similar to NetKAT, Pathetic's semantics gives rise to a derived language model. To see where the language model comes from, note the isomorphism

$$[\![\phi]\!] \in \mathcal{PK} \to 2^{\mathtt{Sw}^*} \cong \mathcal{PK} \to \mathtt{Sw}^* \to 2 \cong \mathcal{PK} \times \mathtt{Sw}^* \to 2 \cong 2^{\mathcal{PK} \times \mathtt{Sw}^*}$$

.

Language model

$$
\begin{aligned}
G(P) &\subseteq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \\
G(\epsilon) &\triangleq \{S{\cdot}\pi_S \mid S \in \mathtt{Sw}\} \\
G(\emptyset) &\triangleq \{\} \\
G(S) &\triangleq \{S'{\cdot}\pi_S{\cdot}\mathsf{dup}{\cdot}\pi_S \mid S' \in \mathtt{Sw}\} \\
G(\star) &\triangleq \{S{\cdot}\pi_{S'}{\cdot}\mathsf{dup}{\cdot}\pi_{S'} \mid S, S' \in \mathtt{Sw}\} \\
G(\overline{P}) &\triangleq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \setminus G(P) \\
G(P.P') &\triangleq G(P) \diamond G(P') \\
G(P|P') &\triangleq G(P) \cup G(P') \\
G(P \cap P') &\triangleq G(P) \cap G(P') \\
G(P^*) &\triangleq \textstyle\bigcup_{n \geq 0} F^n \\
&\text{where } F^0 = \{S{\cdot}S'\} \\
&\text{and } F^{i+1} = G(P) \diamond F^i
\end{aligned}
$$

$$
\begin{aligned}
G(a \Rightarrow P) &\triangleq G(a) \diamond G(P) \\
G(\phi_1 \uplus \phi_2) &\triangleq G(\phi_1) \cup G(\phi_2) \\
G(\phi_1 \pitchfork \phi_2) &\triangleq G(\phi_1) \cap G(\phi_2)
\end{aligned}
$$

Figure 3.4: Pathetic language model

Recall the language model of NetKAT of reduced strings of the form $\mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$, where At is the set of complete tests on packets (*i.e.* predicates that match exactly one packet); P is the set of complete assignments on packets (*i.e.* a sequence of modifications on packet headers such that all output packets are the same, regardless of the input packet); dup is the singleton set containing dup; $A \cdot B$ denotes string concatenation (lifted to sets of strings); and A^* denotes $\cup_i A^i$, where $A^{i+1} \triangleq A \cdot A^i$ and $A^0 \triangleq \{\epsilon\}$ for ϵ the empty string.

We can define a projection from the language model of NetKAT ($\mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$) to $\mathcal{PK} \times \mathtt{Sw}^*$ by noting that the set of packets is isomorphic to the set of complete packet tests ($\mathcal{PK} \cong \mathsf{At}$), projecting out the non-switch header values of each P, and dropping dup. Conversely, we can lift an element of $2^{\mathcal{PK} \times \mathtt{Sw}^*}$ to a subset of $\mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$ by extending each switch S in a path to a complete assignment (where the switch field in the assignment is set to S), and taking the union over every possible such extension.

To illustrate this construction, consider the case where we have only one header field: *tpDst* (which takes values 0 or 1), and the switch field *pt*. Here, the element of the NetKAT language model $\{(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 0) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 0)\}$ would get mapped to $\{tpDst = 0 \times (\mathsf{T} \cdot \mathsf{U})\}$. Similarly, this element gets mapped back into the NetKAT model as

$$\begin{aligned}\{&(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 0) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 0),\\ &(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 1) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 0),\\ &(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 0) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 1),\\ &(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 1) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 1)\}\end{aligned}$$

The resulting language model for Pathetic is shown in Figure 3.4. Because path expressions only operate on the *sw* header of packets, we elide the other fields, and implicitly perform the extension described above. S denotes the test $sw = \mathsf{S}$, and π_S the matching assignment $sw \leftarrow \mathsf{S}$.

Now that Pathetic and NetKAT have a common semantic domain, we can define what it means for a NetKAT program to satisfy a specification. Intuitively, we want this to be true iff every possible path the program forwards packets on is allowed by ϕ:

Definition 1. *p satisfies ϕ, written $p \vDash \phi$, iff $G(p) \subseteq [\![\phi]\!]$, where $G(p)$ is the language model of NetKAT (see Figure 2.3).*

3.4 NetKAT($-, \cap$)

Rather than develop a one-off verifier for Pathetic and NetKAT, we extend NetKAT to a richer language NetKAT($-, \cap$) that we can translate Pathetic into, and then implement

an equivalence checker for NetKAT$(-, \cap)$. We then use this equivalence checker to verify satisfaction.

Equivalence checking A tool that checks equivalence is a powerful basis for verification. It is well-known in language theory that many useful problems can be reduced to equivalence checking. To convince the reader of the utility of this approach, we outline a couple of examples here.

The Pathetic language is useful for restricting the valid paths a packet can take, but sometimes a simpler specification such as *connectivity* is desired: every host in the network should be able to communicate with every other host. As Foster *et al.* showed in [25], connectivity of a policy p can be specified and verified directly in NetKAT itself with an equivalence checker. Because connectivity does not care about specific paths (only that a path exists), we first replace all instances of dup in p with $\mathsf{1}$. This gives us a policy with the same connectivity behavior, but where packets "magically appear" at their destination with no record of the path (history) taken through the network. We write this mapping $\Phi(p)$. Then, we check its equivalence against a term encoding the end-to-end forwarding behavior of a connected network:

$$\Phi(p) \equiv \sum_{(sw,sw',pt,pt')} \begin{pmatrix} switch = sw \cdot port = pt \cdot \\ switch \leftarrow sw' \cdot port \leftarrow pt' \end{pmatrix}$$

where (sw, pt) and (sw', pt') range over all host-facing ports in the network. One advantage of performing this analysis directly in NetKAT is that both terms are dup-free, which leads to a much more efficient verification than checking the full program (why this is true will become clear in later sections).

Similarly, we can use equivalence checking to implement translation validation and check the correctness of the output of a compiler. The NetKAT compiler translates NetKAT terms

into a sequence of OpenFlow forwarding rules for each switch. These rules can be directly encoded back into NetKAT as a cascade of conditional rules:

$$\begin{aligned} c \;=\; & \mathsf{if}\ pat_1\ \mathsf{then}\ acts_1\ \mathsf{else} \\ & \ldots \\ & \mathsf{if}\ pat_k\ \mathsf{then}\ acts_k\ \mathsf{else}\ \mathsf{0} \end{aligned}$$

where each pat_i is a positive conjunction of tests and each act_i is a sequence of modifications. To verify equivalence, we can check if $p \equiv (c \cdot t)^*$, where t is a term encoding the topology of the network.

For more examples of encodings of NetKAT verification problems that use equivalence checking, see Foster *et al.* [25].

3.4.1 NetKAT($-, \cap$) syntax and semantics

To translate Pathetic, we have extended NetKAT with complement ($\overline{p}$) and intersection ($p \cap q$), as shown in Figure 3.5a. We call this extended language NetKAT($-, \cap$). The semantics (Figure 3.5b) of the two new operators is the obvious interpretation, except for the denotation of complement in the history semantics. Instead of defining $[\![\overline{p}]\!]\ \pi{::}h$ as the complement of $[\![p]\!]\ \pi{::}h$ with respect to the full set of histories, it is instead the complement with respect to all histories with the same past h as $\pi{::}h$, $\mathcal{H}\ |_h$. The reason for this change becomes clear when you consider the relation between the history semantics and the language model: when you sequentially compose two terms, the second term does not get to "rewrite" history in the language model (it can only extend or discard it).

Once we have defined NetKAT($-, \cap$), the translation from Pathetic into NetKAT($-, \cap$) (written $(\!|\phi|\!)$ and shown in Figure 3.6) is fairly straightforward.

$$
\begin{array}{rrll}
\text{Fields} & f & ::= & f_1 \mid \cdots \mid f_k \\
\text{Predicates} & a, b & ::= & \mathsf{1} \qquad \textit{Identity} \\
& & \mid & \mathsf{0} \qquad \textit{Drop} \\
& & \mid & f = n \qquad \textit{Test} \\
& & \mid & a + b \qquad \textit{Disjunction} \\
& & \mid & a \cdot b \qquad \textit{Conjunction} \\
& & \mid & \neg a \qquad \textit{Negation} \\
\text{Policies} & p, q & ::= & a \qquad \textit{Filter} \\
& & \mid & f \leftarrow n \qquad \textit{Modification} \\
& & \mid & \overline{p} \qquad \textit{Complement} \\
& & \mid & p \cdot q \qquad \textit{Sequence} \\
& & \mid & p + q \qquad \textit{Union} \\
& & \mid & p \cap q \qquad \textit{Intersection} \\
& & \mid & p^* \qquad \textit{Kleene star} \\
& & \mid & \mathsf{dup} \qquad \textit{Duplication}
\end{array}
$$

(a) NetKAT$(-, \cap)$ syntax.

$$
\begin{aligned}
[\![p]\!] &\in \mathcal{H} \to \mathcal{P}(\mathcal{H}) \\
[\![\mathsf{1}]\!]\ h &\triangleq \{h\} \\
[\![\mathsf{0}]\!]\ h &\triangleq \emptyset \\
[\![f = n]\!]\ \pi{::}h &\triangleq \begin{cases} \{\pi{::}h\} & \text{if } \pi.f = n \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\neg a]\!]\ h &\triangleq \{h\} \setminus ([\![a]\!]\ h) \\
[\![f \leftarrow n]\!]\ \pi{::}h &\triangleq \{\pi[f \mapsto n]{::}h\} \\
[\![\overline{p}]\!]\ \pi{::}h &\triangleq \mathcal{H}\mid_h \setminus [\![p]\!]\ \pi{::}h \\
[\![p \cdot q]\!]\ h &\triangleq ([\![p]\!] \bullet [\![q]\!])\ h \\
[\![p + q]\!]\ h &\triangleq [\![p]\!]\ h \cup [\![q]\!]\ h \\
[\![p \cap q]\!]\ h &\triangleq [\![p]\!]\ h \cap [\![q]\!]\ h \\
[\![p^*]\!]\ h &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} F^i\ h \\
\text{where } F^0\ h &\triangleq \{h\} \text{ and } F^{i+1}\ h \triangleq ([\![p]\!] \bullet F^i)\ h \\
[\![\mathsf{dup}]\!]\ (\pi{::}h) &\triangleq \{\pi{::}(\pi{::}h)\}
\end{aligned}
$$

(b) NetKAT$(-, \cap)$ semantics.

Figure 3.5: NetKAT$(-, \cap)$.

Theorem 1 (Equivalence of NetKAT translation)**.** *For every Pathetic program ϕ, $G(\phi) = G(\langle\!|\phi|\!\rangle)$*

Theorem 2. *$p \vDash \phi$ iff $\langle\!|\phi|\!\rangle \cap \overline{p} \equiv \mathsf{0}$.*

Corollary 1. *$p \vDash \phi$ iff $p \leq \langle\!|\phi|\!\rangle$.*

3.4.2 NetKAT$(-, \cap)$ equational theory

In this section, we extend the equational theory of NetKAT to NetKAT$(-, \cap)$, and prove our axioms complete for a restricted subset. The only content of this section that the reader should know to understand the rest of the thesis are the axioms themselves, presented in Figure 3.9. The rest of the section is not essential to understand this chapter, and can safely

$$
\begin{aligned}
(\!|\epsilon|\!) &\triangleq 1 \\
(\!|\emptyset|\!) &\triangleq 0 \\
(\!|S|\!) &\triangleq sw \leftarrow S \cdot \mathsf{dup} \\
(\!|\star|\!) &\triangleq \textstyle\sum_{S' \in \mathtt{Sw}} sw \leftarrow S' \cdot \mathsf{dup} \\
(\!|\overline{P}|\!) &\triangleq \overline{(\!|P|\!)} \\
(\!|P.P'|\!) &\triangleq (\!|P|\!) \cdot (\!|P'|\!) \\
(\!|(P|P')|\!) &\triangleq (\!|P|\!) + (\!|P'|\!) \\
(\!|(P \cap P')|\!) &\triangleq [\![P]\!] \cap [\![P']\!] \\
(\!|P^*|\!) &\triangleq (\!|P|\!)^* \\
(\!|pr \Rightarrow P|\!) &\triangleq pr \cdot (\!|P|\!) \\
(\!|\phi_1 \uplus \phi_2|\!) &\triangleq (\!|\phi_1|\!) + (\!|\phi_2|\!) \\
(\!|\phi_1 \pitchfork \phi_2|\!) &\triangleq (\!|\phi_1|\!) \cap (\!|\phi_2|\!)
\end{aligned}
$$

Figure 3.6: Translation from Pathetic into NetKAT$(-, \cap)$

Packet semantics

$$
\begin{aligned}
[\![p]\!]_{-\mathsf{dup}} &\in \mathcal{PK} \to \mathcal{P}(\mathcal{PK}) \\
[\![1]\!]_{-\mathsf{dup}}\ \pi &\triangleq \{\pi\} \\
[\![0]\!]_{-\mathsf{dup}}\ \pi &\triangleq \emptyset \\
[\![f = n]\!]_{-\mathsf{dup}}\ \pi &\triangleq \begin{cases} \{\pi\} & \text{if } \pi.f = n \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\neg a]\!]_{-\mathsf{dup}}\ \pi &\triangleq \{\pi\} \setminus ([\![a]\!]_{-\mathsf{dup}}\ \pi) \\
[\![f \leftarrow n]\!]_{-\mathsf{dup}}\ \pi &\triangleq \{\pi[f \mapsto n]\} \\
[\![\overline{p}]\!]_{-\mathsf{dup}}\ \pi &\triangleq \mathcal{PK} \setminus [\![p]\!]_{-\mathsf{dup}}\ \pi \\
[\![p \cdot q]\!]_{-\mathsf{dup}}\ \pi &\triangleq ([\![p]\!]_{-\mathsf{dup}} \bullet [\![q]\!]_{-\mathsf{dup}})\ \pi \\
[\![p + q]\!]_{-\mathsf{dup}}\ \pi &\triangleq [\![p]\!]_{-\mathsf{dup}}\ \pi \cup [\![q]\!]_{-\mathsf{dup}}\ \pi \\
[\![p \cap q]\!]_{-\mathsf{dup}}\ \pi &\triangleq [\![p]\!]_{-\mathsf{dup}}\ \pi \cap [\![q]\!]_{-\mathsf{dup}}\ \pi \\
[\![p^*]\!]_{-\mathsf{dup}}\ \pi &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} F^i\ \pi \\
\text{where } F^0\ \pi \triangleq \{\pi\} &\text{ and } F^{i+1}\ \pi \triangleq ([\![p]\!]_{-\mathsf{dup}} \bullet F^i)\ \pi
\end{aligned}
$$

dup-free Language model

$$
\begin{aligned}
G_{-\mathsf{dup}}(p) &\subseteq \mathsf{At} \cdot P \\
G_{-\mathsf{dup}}(\alpha) &\triangleq \{\alpha \cdot \pi_\alpha\} \\
G_{-\mathsf{dup}}(\pi) &\triangleq \{\alpha \cdot \pi \mid \alpha \in \mathsf{At}\} \\
G_{-\mathsf{dup}}(\overline{p}) &\triangleq \mathsf{At} \cdot P \setminus G_{-\mathsf{dup}}(p) \\
G_{-\mathsf{dup}}(p \cdot q) &\triangleq G_{-\mathsf{dup}}(p) \diamond G_{-\mathsf{dup}}(q) \\
G_{-\mathsf{dup}}(p + q) &\triangleq G_{-\mathsf{dup}}(p) \cup G_{-\mathsf{dup}}(q) \\
G_{-\mathsf{dup}}(p \cap q) &\triangleq G_{-\mathsf{dup}}(p) \cap G_{-\mathsf{dup}}(q) \\
G_{-\mathsf{dup}}(p^*) &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} G_{-\mathsf{dup}}(p^i)
\end{aligned}
$$

Figure 3.7: NetKAT$(-, \cap)$ dup-free semantics and language model.

Language model

$$
\begin{aligned}
G(p) &\subseteq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \\
G(\alpha) &\triangleq \{\alpha{\cdot}\pi_\alpha\} \\
G(\pi) &\triangleq \{\alpha{\cdot}\pi \mid \alpha \in \mathsf{At}\} \\
G(\overline{p}) &\triangleq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \setminus G(p) \\
G(p \cdot q) &\triangleq G(p) \diamond G(q) \\
G(p + q) &\triangleq G(p) \cup G(q) \\
G(p \cap q) &\triangleq G(p) \cap G(q) \\
G(\mathsf{dup}) &\triangleq \{\alpha{\cdot}\pi_\alpha{\cdot}\mathsf{dup}{\cdot}\pi_\alpha \mid \alpha \in \mathsf{At}\} \\
G(p^*) &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} G(p^i)
\end{aligned}
$$

Figure 3.8: NetKAT$(-, \cap)$ language model.

be skipped by the reader.

The original NetKAT language has a sound and complete equational theory, based on the equational theory of Kleene Algebra with test (KAT) [56]. Because NetKAT$(-, \cap)$ is a conservative extension of NetKAT, the NetKAT equational theory is sound, but not complete for NetKAT$(-, \cap)$. Unfortunately, we conjecture that there is no finite extension of the NetKAT axioms that is sound and complete for NetKAT$(-, \cap)$ (Conjecture 1).

Instead, in this section we give a sound and complete axiomatization of the dup-free fragment of NetKAT$(-, \cap)$, as shown in Figure 3.9. The semantics of dup-free NetKAT$(-, \cap)$ is given by a semantics over packets, instead of histories, and is shown in Figure 3.7.

For this axiomatization, we use α, β as short-hand to denote *complete tests*, a conjunction of header tests $f = n$ such that every header f appears exactly once in a specific order. Similarly, we use π, γ to denote *complete assignments*, a sequence of header modifications $f \leftarrow n$ such that every header f appears exactly once. We write α_π to denote the complete test corresponding to π (*i.e.* the test that matches exactly the packet containing the header

$$
\begin{aligned}
\mathsf{all} &\triangleq \textstyle\sum_\beta \pi_\beta && \\
p \cap q &\equiv q \cap p && \text{INTER-COMM} \\
p \cap (q \cap r) &\equiv (q \cap p) \cap r && \text{INTER-ASSOC} \\
p \cap p &\equiv p && \text{INTER-IDEM} \\
a \cap b &\equiv a \cdot b && \text{INTER-FILTER} \\
f \leftarrow n \cap a &\equiv f = n \cdot a && \text{INTER-MOD-FILTER} \\
f \leftarrow n \cap f' \leftarrow n' &\equiv (f \leftarrow n \cdot f' = n') + (f' \leftarrow n' \cdot f = n) && \text{INTER-MOD-MOD} \\
(p + q) \cap r &\equiv (p \cap r) + (q \cap r) && \text{INTER-PAR-DIST} \\
(p \cap q) + r &\equiv (p + r) \cap (q + r) && \text{PAR-INTER-DIST} \\
(a \cdot p) \cap q &\equiv a \cdot (p \cap q) && \text{INTER-FILTER-DIST-LEFT} \\
(p \cdot a) \cap q &\equiv (p \cap q) \cdot a && \text{INTER-FILTER-DIST-RIGHT} \\
f \leftarrow n \cdot (p \cap q) &\equiv (f \leftarrow n \cdot p) \cap (f \leftarrow n \cdot q) && \text{INTER-MOD-DIST-LEFT} \\
\overline{f = n} &\equiv \neg f = n \cdot \textstyle\sum_\pi \pi + \sum_\alpha \sum_{\pi \neq \pi_\alpha} \alpha \cdot \pi && \text{COMP-FILTER}^* \\
\overline{f \leftarrow n} &\equiv \textstyle\sum_\alpha \sum_{\pi \neq \pi_{\alpha[f \leftarrow n]}} \alpha \cdot \pi && \text{COMP-MOD}^* \\
\overline{p + q} &\equiv \overline{p} \cap \overline{q} && \text{COMP-PAR} \\
\overline{p \cap q} &\equiv \overline{p} + \overline{q} && \text{COMP-INTER} \\
\overline{p \cdot q} &\equiv \textstyle\bigcap_\gamma (\overline{p} \cdot \gamma \cdot \mathsf{all} + \pi_\gamma \cdot \overline{q}) && \text{COMP-SEQ}^*
\end{aligned}
$$

Figure 3.9: NetKAT$(-, \cap)$ Axioms. Axioms labeled with * are only valid in the dup-free fragment

values set by π), and π_α to denote the complete assignment corresponding to α. Note that because the set of headers and the set of header values are both finite, sums over the set of complete tests and complete assignments are also finite, and thus valid short-hand for the axioms.

Most of the new axioms for complement and intersection (Figure 3.9) are self-explanatory, except for the axiom for the complement of a sequential composition, COMP-SEQ. To understand the axiom, first note that the abbreviation all is a unit for intersection, and an annihilator for parallel composition in the dup-free fragment. *i.e.* $p \cap \mathsf{all} \equiv p$ and $p + \mathsf{all} \equiv \mathsf{all}$. Thus, the $p \cdot \mathsf{all} + q$ essentially says "if p is non-zero, ignore q". Looking at the semantics of

the complement of a sequential composition:

$$\begin{aligned}\llbracket\overline{p \cdot q}\rrbracket_{-\mathsf{dup}}\ pk &= \overline{\bigcup_{pk' \in \llbracket p\rrbracket_{-\mathsf{dup}}\ pk} \llbracket q\rrbracket_{-\mathsf{dup}}\ pk'} \\ &= \bigcap_{pk' \in \llbracket p\rrbracket_{-\mathsf{dup}}\ pk} \llbracket\overline{q}\rrbracket_{-\mathsf{dup}}\ pk'\end{aligned}$$

So, this is equivalent to "guessing" each pk' in $\llbracket p\rrbracket_{-\mathsf{dup}}\ pk$, computing $\llbracket\overline{q}\rrbracket_{-\mathsf{dup}}\ pk'$, ignoring the result if we "guessed wrong" (i.e. $pk' \in \llbracket\overline{p}\rrbracket_{-\mathsf{dup}}\ pk$), and taking the intersection over all such guesses.

This trick works for the dup-free fragment because we can filter out the "wrong guesses" by testing against $\overline{p}$. However, it's not at all clear how to lift this trick to the history semantics: because terms ignore all but the last packet in the history, we can't create a term that filters out "wrong guesses in the past".

Theorem 3 (dup-free Soundness of NetKAT$(-,\cap)$ Axioms)**.** *For all dup-free policies p and q, if $p \equiv q$ is provable from the NetKAT$(-,\cap)$ axioms, then $\llbracket p\rrbracket_{-\mathsf{dup}} = \llbracket q\rrbracket_{-\mathsf{dup}}$.*

Theorem 4 (Soundness of non-complement axioms)**.** *For all policies p and q, if*

$$p \equiv q$$

in the equational theory generated by the NetKAT$(-,\cap)$ axioms minus COMP-FILTER*,* COMP-MOD*, and* COMP-SEQ*, then*

$$\llbracket p\rrbracket = \llbracket q\rrbracket$$

dup-free completeness The original proof of NetKAT completeness defined a restricted subset of NetKAT, reduced NetKAT, and showed that every NetKAT term was provably equivalent to a reduced NetKAT term. Next they gave a language model for reduced NetKAT

that is isomorphic to the standard (history) semantics. Finally, they defined a normal form for reduced NetKAT and related it to regular sets of guarded string, and showed that every reduced NetKAT policy is provably equivalent to a policy in normal form. Completeness then follows as a corollary of the completeness of KAT.

To prove completeness for NetKAT$(-,\cap)$, we can carry out a parallel development for dup-free NetKAT. We then show that we can translate any term in the dup-free fragment of NetKAT$(-,\cap)$ language into a provably equivalent term in reduced dup-free NetKAT. Completeness then follows as an immediate corollary of the completeness of NetKAT itself.[4]

Lemma 1. *Every dup-free NetKAT$(-,\cap)$ policy is provably equivalent to a reduced NetKAT$(-,\cap)$ policy.*

Theorem 5. *The axioms for NetKAT$(-,\cap)$ shown in Figure 3.9 plus the NetKAT axioms (minus* PA-DUP-FILTER-COMM*) are complete for the dup-free fragment.*

Conjecture 1. *There does not exist any sound, finite equational extension of the NetKAT axioms that is complete for NetKAT$(-,\cap)$.*

3.5 NetKAT$(-,\cap)$ automata theory

At this point, we have presented a specification language Pathetic, defined what it means for a NetKAT policy to satisfy a specification, and extended NetKAT to NetKAT$(-,\cap)$ in order to translate Pathetic.

[4]At first glance, it might seem that we are missing axioms. For example, we have no axiom explaining how to take the complement of p^*. It turns out that such an axiom is unnecessary for the dup-free fragment. Because dup-free NetKAT$(-,\cap)$ is essentially finite, any instance of star can be transformed into an equivalent star-free term.

The axioms and the proof are closely tied to the fact that we are working in the dup-free fragment. To extend the development to the full language would require new axioms for complement, including an axiomatization of $\overline{p^*}$.

From here, the road map is as follows: (1) we review the theory of NetKAT$(-,\cap)$ coalgebras, which form the foundation for NetKAT$(-,\cap)$ automata; (2) we show how to construct (deterministic) NetKAT$(-,\cap)$ automata from terms; (3) we describe a bisimulation checker for NetKAT$(-,\cap)$ automata. We then appeal to the fact that bisimilarity of NetKAT$(-,\cap)$ automata corresponds to language equivalence, and we are done building an equivalence checker.

3.5.1 NetKAT$(-,\cap)$ coalgebra

In this section, we briefly review the coalgebraic theory of NetKAT$(-,\cap)$, and in the next section show how to use it to build to build an automata-theoretic equivalence checker. This section is essentially a recapitulation of the development in Foster *et al.* [25], and we include it for the sake of completeness. Readers familiar with that paper can safely skip reading this section.

In this thesis, we take the coalgebraic view of automata, where an automaton is simply a finite-state coalgebra over a state space S, along with an observation map $S \to 2$ indicating accepting states, and a continuation map $S \times \Sigma \to S$ specifying state transitions. For more details on this approach to representing state-based transition systems, see Rutten [89].

Concretely, a NetKAT coalgebra consists of a state space S, along with observation and continuation maps

$$\epsilon_{\alpha\beta} : S \to 2 \qquad\qquad \delta_{\alpha\beta} : S \to S$$

for $\alpha, \beta \in \mathsf{At}$. A deterministic NetKAT automaton is a finite-state NetKAT coalgebra with a start state in S. The automaton operates on the (reduced) strings of the language-model, $U = \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$. If the automaton is in state s, and sees string $\alpha\pi_\beta$, then it accepts iff $\epsilon_{\alpha\beta}(s)$. If the automaton is in state s and sees string $\alpha{\cdot}\pi_\beta{\cdot}\mathsf{dup}{\cdot}x$, then it transitions to $\delta_{\alpha\beta}(s)$ with residual string $\beta{\cdot}x$.

A reduced string is accepted by the automaton iff it accepts the string from the distinguished start state.

As Foster *et al.* [25] showed, NetKAT's automata theory has an analog to Kleene's theorem for regular expressions:

Theorem 6 (Kleene's Theorem for NetKAT)**.** *A set of string is $G(p)$ for some NetKAT policy p iff it is the set of strings accepted by some finite NetKAT automaton.*

For the full details of this theorem and its proof, read [25].

3.5.2 NetKAT$(-, \cap)$ automata

Because NetKAT automata are closed under intersection and complement, it immediately follows that NetKAT$(-, \cap)$ automata are in fact NetKAT automata. However, this does not mean that we can just reuse the NetKAT automata construction. Foster *et al.* also showed that the size of the minimal deterministic automata M_p for each NetKAT term p is $O(2^l)$, where l is the number of occurences of dup in p. It is well-known [95] that enriching regular expressions with complement and intersection causes at least an exponential increase in the size of the minimal automata [31], to doubly-exponential. This lower-bound also applies to our language. Therefore, their construction cannot apply to NetKAT$(-, \cap)$.

Despite this large theoretical lower-bound, we can still hope to build a verifier that is performant in practice. By carefully only exploring the reachable state space, and generating the automata on demand, we can avoid unnecessary work. Moreover, we expect that real-world specifications will not exhibit the pathological structure that causes such an explosion.

Brzozowski Derivative The Brzozowski derivative is a standard way of building coalgebras from regular expressions. There is both a *semantic* Brzozowski derivative defined upon subsets of U, giving a NetKAT$(-,\cap)$ coalgebra over the state space 2^U, and a *syntactic* Brzozowski derivative defined over NetKAT$(-,\cap)$ expressions, resulting in a NetKAT$(-,\cap)$ coalgebra over a state space of expressions (shown in Figure 3.10).

3.6 Automata Representation

In this section, we show how to build a Pathetic verifier based upon the automata theory in the previous section. We start by outlining the Brzozowski-based construction and representation used by Foster *et al.* for NetKAT verifier, and explain why this representation does not work for NetKAT$(-,\cap)$. We then present our own construction, also based on the Brzozowski derivative.

Foster *et al.*'s automata representation Looking carefully at the definition of the Brzozowski derivatives, it becomes clear that each of the definitions corresponds to operations on $\mathsf{At} \times \mathsf{At}$ matrices, where $E_{\alpha\beta}(p)$ is the α, β entry of the matrix $E(p)$. For example, the definition of $E_{\alpha\beta}(p \cdot q)$ is exactly the definition of matrix multiplication. Moreover, as Foster *et al.* note, for NetKAT, these matrices are highly sparse, and "close" to a diagonal matrix. Consider, for example, $E_{\alpha\beta}(f = n)$. This corresponds to a diagonal binary matrix where

$$D'_{\alpha\beta}(\pi) = 0 \qquad D'_{\alpha\beta}(b) = 0 \qquad D'_{\alpha\beta}(\mathsf{dup}) = [\alpha = \beta]$$

$$D'_{\alpha\beta}(\overline{p}) = \overline{D'_{\alpha\beta}(p)}$$

$$D'_{\alpha\beta}(p + q) = D'_{\alpha\beta}(p) + D'_{\alpha\beta}(q)$$

$$D'_{\alpha\beta}(p \cdot q) = D'_{\alpha\beta}(p) \cdot q + \sum_{\gamma} E_{\alpha\gamma}(p) \cdot D'_{\gamma\beta}(q)$$

$$D'_{\alpha\beta}(p \cap q) = D'_{\alpha\beta}(p) \cap D'_{\alpha\beta}(q)$$

$$D'_{\alpha\beta}(p^*) = D'_{\alpha\beta}(p) \cdot p^* + \sum_{\gamma} E_{\alpha\gamma}(p) \cdot D'_{\gamma\beta}(p^*)$$

$$D_{\alpha\beta}(p) = \beta \cdot D'_{\alpha\beta}(p)$$

$$E_{\alpha\beta}(\pi) = [\pi = \pi_\beta] \qquad E_{\alpha\beta}(b) = [\alpha = \beta \leq b] \qquad E_{\alpha\beta}(\mathsf{dup}) = 0$$

$$E_{\alpha\beta}(\overline{p}) = \overline{E_{\alpha\beta}(p)} \qquad E_{\alpha\beta}(p + q) = E_{\alpha\beta}(p) + E_{\alpha\beta}(q)$$

$$E_{\alpha\beta}(p \cap q) = E_{\alpha\beta}(p) \cdot E_{\alpha\beta}(q)$$

$$E_{\alpha\beta}(p \cdot q) = \sum_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(q)$$

$$E_{\alpha\beta}(p^*) = [\alpha = \beta] + \sum_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(p^*).$$

Figure 3.10: NetKAT$(-, \cap)$ syntactic Brzozowski derivative.

entry $\alpha\alpha$ is 1 iff the f value of α is n. By using a sparse matrix representation based upon these vectors (and similar vectors corresponding to modifications), they obtain a compact representation that requires much less space than the $|\mathsf{At}|^2$ entries of the full matrix. Note that $\mid \mathsf{At} \mid = \Pi_f \mid v_f \mid$, where v_f is the set of values for header field f, *i.e.* exponential in the number of fields.

Second, they observe that the size of the state space S can be bounded based upon the structure of the term. They identify a set of subterms, dubbed *spines*, whose size is linear in l, the number of occurrences of dup in the original expression, and show that sets of spines

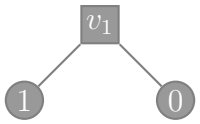
suffice as a representation of S. Note that the linear bound on the set of spines this leads directly to an exponential bound on the size of S: 2^l.

Unfortunately, neither of these tricks will work for NetKAT$(-, \cap)$. The complement operation of NetKAT$(-, \cap)$ takes a sparse matrix to its complement, which is a dense matrix. This eliminates the benefits of their sparse matrix representation. Similarly, their small set of *spines* no longer suffices to represent the state space. This is immediately obvious because using spines leads to an exponential bound on the size of the state space, but minimal NetKAT$(-, \cap)$ automata may be super-exponentially sized.

3.6.1 NetKAT$(-, \cap)$ automata representation

We use a different, novel representation for NetKAT$(-, \cap)$ derivatives that retains the benefits of the sparse representation of Foster *et al.* when possible. We combine an efficient representation of sparse matrices, based on functional decision diagrams, with an algebraic representation of matrix operations. This enables the verifier to exploit sparseness for positive terms (terms without complement), and allows the recognition of simplifying algebraic identities for the full language (*e.g.* $\overline{\overline{p}} \equiv p$, $\overline{p} + \overline{q} \equiv \overline{p \cap q}$).

NetKAT$(-, \cap)$ E derivative representation We use functional decision diagrams (**FDD**s), a generalization of binary decision diagrams, to represent E matrices. A binary decision diagram (BDD) represents a boolean valued function on a set of boolean valued variables $(H \to 2) \to 2$, where H is the set of variables. Concretely, a BDD is a directed, acyclic graph where leaf nodes are labeled with boolean values, and interior nodes are labeled with variables and have two outgoing edges, representing the two possible values of the variable. For example,



is a BDD that represents the boolean function $v_1 = 1$ (we draw the true edge on the left, and the false edge on the right).

Similarly, an (H, V, B) **FDD**, is a directed, acyclic graph that represents a function of type $(H \to V) \to B$, where the variables in H are V valued. We replace the boolean variables with boolean tests of equality on mutli-valued variables. Thus, an (H, V, B) has as internal nodes pairs in $(H \times V)$, representing boolean tests on the input, with children nodes representing the path for inputs that satisfy or fail the test, and its leaf nodes are elements of B. Just as in BDDs, an **FDD** that represents a highly uniform function may have many identical subgraphs. Reduced **FDD**s use structure sharing to eliminate common sub-**FDD**s, and can provide very compact representations for uniform functions in which large sets of inputs have the same output. In this rest of this chapter, **FDD** means a reduced **FDD**.

If E is an (H, V, B) **FDD**, and h is an element of $(H \to V)$, then we write $[\![E]\!](h)$ to mean the element of B that is output by the function represented by E on the input h.

NetKAT **FDD**s (herein just **FDD**s) are $(F, \mathbb{N}, \mathcal{P}(F \rightharpoonup \mathbb{N}))$ **FDD**s. That is, they directly represent functions that take packets as input (represented as finite maps from headers f to header values n), and output sets of (partial) packets (finite maps that may not have values for all headers). We interpret a NetKAT **FDD** E as a function of type $\alpha \to \beta \to 2$ by $E(\alpha)(\beta) = \exists \beta' \in [\![F]\!](\alpha) \wedge \alpha \circ \beta' = \beta$. For example, the **FDD** $\{[]\}$ corresponds to the E derivative of the diagonal 1: $\lambda\alpha, \beta.\alpha = \beta$.

Similarly, the **FDD** representation for $E(f \leftarrow n)$ is just: $\{[f = n]\}$.

$$\textbf{FDD}\quad E, E' ::= \{f \rightharpoonup n\} \mid (f = n)?(E) : (E')$$

$$[\![E]\!] \in \mathsf{At} \to \mathsf{At} \to 2$$

$$[\![\{m_i\}]\!](\alpha)(\beta) \triangleq \textstyle\sum_{m_i} [(\alpha \circ m_i) = \beta]$$

$$[\![(f = n)?(E) : (E')]\!] \triangleq \begin{cases} [\![E]\!](\alpha)(\beta) & \text{if } \alpha[f] = n \\ [\![E']\!](\alpha)(\beta) & \text{o.w.} \end{cases}$$

Figure 3.11: NetKAT **FDD** syntax and semantics

NetKAT FDD operations

$$E_{f \leftarrow n} \triangleq \{[f \leftarrow n]\}$$

$$E_{f=n} \triangleq (f = n)?(\{[]\}) : (\{\})$$

$$E_{p+q} \triangleq E_p \cup E_q$$

$$E_{p \cap q} \triangleq E_p \cap E_q$$

$$E_{p \cdot q} \triangleq E_p \cdot E_q$$

$$E_{p^*} \triangleq \mu E'.E_1 + E_p \cdot E'$$

The basic operations on **FDD**s are union, intersection, sequential composition, iteration.

The **FDD** representation is an alternative to the sparse matrix representation of Foster *et al.*, and suffers from the same problem when applied to NetKAT$(-, \cap)$. Notice that the **FDD** representation of the term 1 is very compact. This representation is optimized for sparse matrices that are close to a diagonal matrix. Consider, by contrast, the representation of the E derivative of term $\overline{1}$. It is equivalent to $\lambda\alpha, \beta.\alpha \neq \beta$. But this function has a very large representation as an **FDD**: it is the full tree where every full path through the tree represents a complete test, and the leaf node on the path corresponding to α is the set $\{\beta \mid \beta \neq \alpha\}$. Even worse, a naive implementation may end up constructing a very large **FDD** as an intermediate state when the final **FDD** is in fact very small. For example, naively constructing the **FDD** for $\overline{\overline{1}} \equiv 1$ would result in an **FDD** equivalent to $\{[]\}$, but

construct the **FDD** for $\overline{1}$ as an intermediate.

To avoid the blow-up that comes from complementing **FDD**s, but maintain the benefit of their compactness when possible, we combine **FDD**s with the symbolic representation shown in Figure 3.6.1. Complement-free policies are represented as **FDD**s (smart constructors enforce that the union, intersection, iteration, and sequential composition of **FDD**s are **FDD**s), and policies containing complement are represented as formal terms over **FDD**s. This enables compact representation of positive NetKAT terms while enabling the recognition of simplifying algebraic identities. In this representation, the term $\overline{\overline{1}}$ would be represented exactly as the **FDD** $\{[]\}$, because the symbolic identity $\overline{\overline{p}} \equiv p$ would be recognized and reduced, without computing the **FDD** for $\overline{p}$.

We write E_p to refer to the **FDD** representation of the E derivative of the (complement-free) term p.

NetKAT$(-, \cap)$ derivative representation

E	e, e'	$::= E$	*Positive **FDD***
		$\mid \overline{e}$	*Complement*
		$\mid e + e'$	*Union*
		$\mid e \cap e'$	*Intersection*
		$\mid e \cdot e'$	*Sequential composition*
		$\mid e^*$	*Kleene star*

NetKAT($-, \cap$) derivative representation semantics

$$
\begin{aligned}
[\![e]\!] &\in \mathsf{At} \to \mathsf{At} \to 2 \\
[\![E]\!](\alpha)(\beta) &\triangleq [\![E]\!](\alpha)(\beta) \\
[\![\bar{e}]\!](\alpha)(\beta) &\triangleq \overline{[\![e]\!](\alpha)(\beta)} \\
[\![e + e']\!](\alpha)(\beta) &\triangleq [\![e]\!](\alpha)(\beta) + [\![e']\!](\alpha)(\beta) \\
[\![e \cap e']\!](\alpha)(\beta) &\triangleq [\![e]\!](\alpha)(\beta) \cdot [\![e']\!](\alpha)(\beta) \\
[\![e \cdot e']\!](\alpha)(\beta) &\triangleq \textstyle\sum_\gamma [\![e]\!](\alpha)(\gamma) \cdot [\![e']\!](\gamma)(\beta) \\
[\![e^*]\!](\alpha)(\beta) &\triangleq [\alpha = \beta] + \textstyle\sum_\gamma [\![e]\!](\alpha)(\gamma) \cdot [\![e^*]\!](\alpha)(\beta)
\end{aligned}
$$

To represent D derivatives, we follow the insight of Foster *et al.* that the D derivatives correspond to basic matrix operations, and use a matrix-like representation. We decompose each D derivative into a sum of (possibly overlapping) single-valued matrices, and represent each matrix as a pair of an E matrix (representing the domain), and a policy (representing the value of the matrix on its domain) (shown in Figure 3.12)[5]. The relationship between this representation and the syntactic Brzozowski derivative is expressed in Lemma 2:

Lemma 2.

$$
D_{\alpha,\beta}(p) \equiv \sum_{(e,d) \in D(p)} [e(\alpha)(\beta)] \cdot \beta \cdot d
$$

3.6.2 NetKAT($-, \cap$) equivalence checking

With our automata representation, we can now build an equivalence checker that checks NetKAT($-, \cap$) terms for bisimulation. Given two NetKAT($-, \cap$) terms p and q, we first compare their E matrices for (semantic) equality. If they are not equal, we return false.

[5]This representation is based upon one proposed by Konstantinos Mamouras in private correspondence.

$$E(p) = E_p \qquad E(b) = E_b \qquad E(\mathsf{dup}) = E_0$$
$$E(\overline{e}) = \overline{E(e)} \qquad E(e_1 + e_2) = E(e_1) + E(e_2)$$
$$E(e_1 \cap e_2) = E(e_1) \cap E(e_2)$$
$$E(e_1 \cdot e_2) = E(e_1) \cdot E(e_2)$$
$$E(e^*) = E(e)^*$$

$$D(p) = \{\} \qquad D(b) = \{\} \qquad D(\mathsf{dup}) = \{(E(1), 1)\}$$
$$D(e_1 + e_2) = D(e_1) \cup D(e_2)$$
$$D(e_1 \cdot e_2) = D(e_1) \cdot e_2 \cup E(e_1) \cdot D(e_2)$$
$$D(e_1 \cap e_2) = \{d_1 \cap d_2 \mid d_1 \in D(e_1), d_2 \in D(e_2)\}$$
$$D(e^*) = E(e^*) \cdot D(e) \cdot e^*$$
$$D(\overline{p}) = \bigcup_{\alpha,\beta} \left\{ (E(\alpha \cdot p_\beta), \bigcap_{(e',d') \in D(p) \wedge e'(\alpha)(\beta)} \overline{d'}) \right\}$$

where

$$D \cdot p \triangleq \{(e, d \cdot p) \mid (e, d) \in D\}$$
$$E \cdot D \triangleq \{(E \cdot e, d) \mid (e, d) \in D\}$$
$$(e, d) \cap (e', d') \triangleq (e \cap e', d \cap d')$$

Figure 3.12: NetKAT$(-, \cap)$ derivative representation.

Otherwise, we calculate the derivatives of both terms and recursively check them for equivalence. Once we've reached every reachable pair of derivatives, the algorithm halts. The proof of termination depends upon finiteness of an extension of the original NetKAT spines to NetKAT$(-, \cap)$, and the closure of these spines under the derivative, which is shown in Appendix A.1.

CHAPTER 4

CORRECTLY IMPLEMENTING NETWORK PROGRAMS

"Trust, but verify."

—Ronald Reagan

In the previous chapter we showed how to verify that a network program correctly implements a specification. In this chapter, we show how to build a system that correctly implements network programs, guaranteeing that the properties of the input program also are preserved by the resulting network itself.

Concretely, this chapter describes the design and implementation of a machine-verified compiler and OpenFlow controller for the NetCore language, a predecessor to NetKAT. Starting from the foundations, we develop a detailed operational model for the OpenFlow SDN platform, and formalize it in the Coq proof assistant. We then use this model to develop a verified compiler and run-time system for a high-level network programming language (NetCore). We identify bugs in existing languages and tools built without formal foundations, and prove that these bugs are absent from our system. Finally, we describe our prototype implementation and our experiences using it to build practical applications.

The content of this chapter is based upon a joint PLDI paper [35] published with Arjun Guha and Nate Foster in 2013.

4.1 Introduction

Bugs in compilers and runtimes are especially pernicious sources of errors. Difficult to track down, their effect can be widespread, potentially affecting every program they touch.

Indeed, a lack of trust in the reliability of complex optimizing compilers and language runtime systems is one potential stumbling block in the adoption of high-level programming languages in the systems domain.

Moreover, recent work has shown that such mistrust would not be entirely misplaced: NICE [14] found a number of runtime bugs in popular SDN controller platforms, and in prior work [35] we found correctness bugs in every network programming language compiler examined.

Fortunately, there is a solution: formal specification and verification of compilers and runtimes. In one study of optimizing C compilers, every single compiler, save one, was found to have bugs that caused wrong-code generation [108]. The one exception was the formally verified compiler from the CompCert project [58][1].

In this chapter, we show how to formally model network programming languages and software-defined networks in the Coq theorem prover. We then show how to use these formal models to build and verify a compiler and network controller that provably preserves the correctness of its input program.

Architecture Our system is organized as a verified software stack (Figure 4.1) that translates through the following levels of abstraction:

- **NetCore.** The highest level of abstraction is the NetCore language, proposed in prior work by Monsanto *et al.* [74]. NetCore is a predecessor to the NetKAT language used earlier in this thesis. Unlike NetKAT, NetCore does not directly model the topology of

[1]Initially, the authors of that study found a bug in an unverified component of CompCert. In response, CompCert extended the verified to include that component, and the authors were not able to find any bugs in the newly verified system.

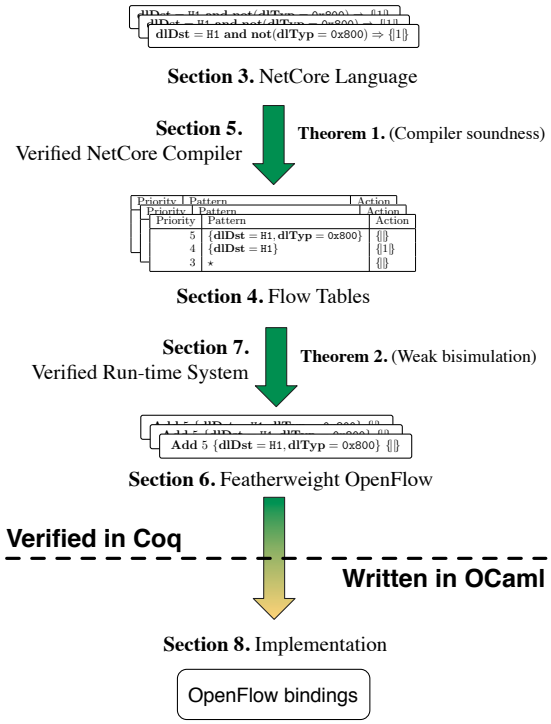


Figure 4.1: System architecture.

the network, and so is essentially equivalent to dup-free NetKAT. NetCore also lacks the iteration operator of NetKAT, but iteration adds no expressivity to the dup-free fragment, so this is not a significant loss.

- **Flow tables.** The intermediate level of abstraction is *flow tables*, a representation that sits between NetCore programs and switch-level configurations. There are two main differences between NetCore programs and flow tables. First, NetCore programs describe the forwarding behavior of a whole network, while flow tables describe the behavior of a single switch. Second, flow tables process packets using a linear scan through a list of prioritized rules. Hence, to translate operators such as union and negation, the NetCore compiler must generate a sequence of rules that encodes the same semantics. However, because flow table matching uses a lower-level packet representation (as nested frames of Ethernet, IP, TCP, etc. packets), flow tables must satisfy

a well-formedness condition to rule out invalid patterns that are inconsistent with this representation.

- **Featherweight OpenFlow.** The lowest level of abstraction is *Featherweight OpenFlow*, a new foundational model we have designed that captures the essential features of SDNs. Featherweight OpenFlow models switches, the controller, the network topology, as well as their internal transitions and interactions in a small-step operational semantics. This semantics is non-deterministic, modeling the asynchrony inherent in networks. To implement a flow table in a Featherweight OpenFlow network, the controller instructs switches to install or uninstall rules as appropriate while dealing with two important issues: First, switches process instructions concurrently with packets flowing through the network, so it must ensure that at all times the rules installed on switches are consistent with the flow table. Second, switches are allowed to buffer instructions and apply them in any order, so it must ensure that the behavior is correct no matter how instructions are reordered through careful use of synchronization primitives.

4.2 Overview

To motivate the need for verified SDN controllers, consider a simplified version of the network from our running example, shown in Figure 4.2. This network has only one switch I, one firewall FW, a load balancer LB, a web server WEB, an internal network INTRANET, and an external network WORLD.

Now imagine we want to build an SDN controller that implements the following network policy: block inbound SSH traffic, route inbound HTTP requests through the load-balancer and then to WEB, and allow all other traffic to INTRANET once it has passed through the

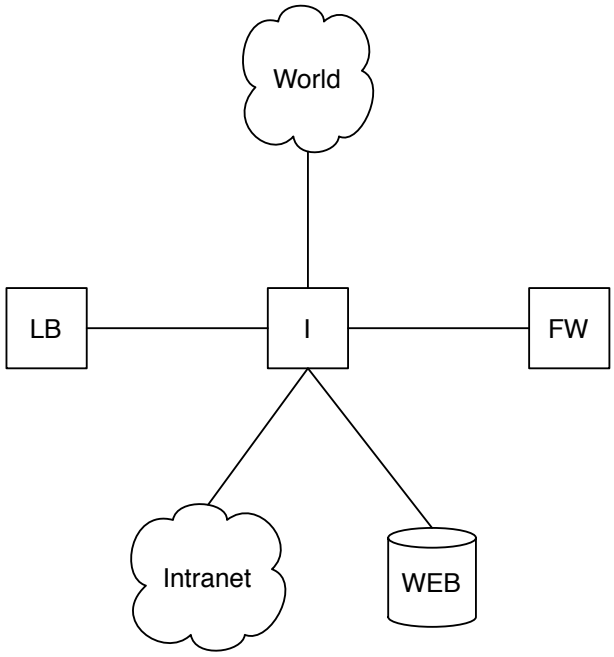


Figure 4.2: Example network topology.

firewall. It is straightforward to formalize this policy as a packet-processing function that maps input packets to (possibly several) output packets: the function drops SSH packets a forwards HTTP packets both to their destination and to the middlebox, and forwards all other packets to the firewall and then their destination.

To implement this function in an SDN, however, we would need to specify several additional low-level details, since switches cannot implement general packet-processing functions directly. First, the controller would need to encode the function as a *flow table*—a set of prioritized forwarding rules. Second, it would need to send the switch a series of control messages to add individual entries from the flow table, incrementally building up the complete table.

More concretely, the controller could first send a message instructing the switch to add

a flow table entry that blocks SSH traffic:

$$\textbf{Add } 10 \ \{\textbf{tpDst} = 22\} \ \{\!|\!\}$$

Here 10 is a priority number, $\{\textbf{tpDst} = 22\}$ is a pattern that matches SSH traffic (TCP port 22), and $\{\!|\!\}$ is an empty multiset of ports, which drops packets, as intended. Next, the controller could add an entry to process inbound HTTP requests:

$$\textbf{Add } 9 \ \{\textbf{inPort} = \mathsf{WORLD}, \textbf{dlDst} = \mathit{WEB}, \textbf{tpDst} = 80\} \ \{\!|\mathsf{LB}|\!\}$$

Note that this rule only applies to HTTP (TCP port 80) packets traffic that has not been sent to LB yet.

The controller can then add another entry to process load-balanced HTTP requests:

$$\textbf{Add } 8 \ \{\textbf{inPort} = \mathsf{LB}, \textbf{dlDst} = \mathit{WEB}, \textbf{tpDst} = 80\} \ \{\!|\mathsf{WEB}|\!\}$$

Finally, the controller could similarly install a pair of entries to forward other packets to their destination, after going through the firewall

$$\textbf{Add } 2 \ \{\textbf{inPort} = \mathsf{WORLD}\} \ \{\!|\mathsf{FW}|\!\}$$

$$\textbf{Add } 1 \ \{\textbf{inPort} = \mathit{FW}\} \ \{\!|\mathsf{INTRANET}|\!\}$$

Note that this rule does not apply to SSH and HTTP traffic, since those packets are handled by the higher-priority rules.

After these control messages have been sent, it would be natural to expect that the network correctly implements the packet-processing function described above. But the situation is actually more complicated: switches have substantial latitude in how they process messages from the controller, and packets may arrive at any time during processing. Establishing that the network correctly implements this function—in particular, that it blocks SSH traffic and load balances HTTP traffic—requires additional reasoning.

Controller-switch consistency. Switches process packets and control messages concurrently. In our example, the switch may receive an HTTP request before the flow table entry that handles HTTP packets arrives. In this case, the switch will send the packet to the controller for further processing. Since the controller is a general-purpose machine, it can implement the packet-processing function directly, apply it to the incoming packet, and send the results back to the switch. However, this means that SDN controllers typically have two *different* implementations of the function: one residing at the controller and another on the switches. A key property we verify is that these two implementations are consistent.

Message reordering. SDN switches may process control messages in any order, and many switches do, to maximize performance. But unrestricted reordering can cause implementations to violate their intended specifications. For example, if the rule to drop SSH traffic is installed after the final, low-priority rule that forwards all traffic, then SSH traffic will temporarily be forwarded by the low-priority rule, breaking the intended security policy. To ensure that such reorderings do not occur, a controller must carefully insert *barrier messages*, which force the switch to process all outstanding messages. A key property we verify is that controllers use barriers correctly (several unverified controllers ignore this issue).

Natural patterns. Another complication is that the patterns presented earlier in this section, such as $\{\mathbf{tpDst} = 22\}$, are actually invalid. To match SSH traffic, it is not enough to simply state that the destination port must be 22. The pattern must also specify that the Ethernet frame type must be IP, and the transport protocol must be TCP. Without these additional constraints, switches will interpret the pattern as a wildcard that matches all packets. Several earlier controller platforms did not properly account for this behavior, and had bugs as a result. We develop a semantics for patterns and identify a class of *natural*

Packet	*pk*	::= **Eth** *dlSrc dlDst dlTyp nwPk*
Network layer	*nwPk*	::= **IP** *nwSrc nwDst nwProto tpPk*
		| **Unknown** *payload*
Transport layer	*tpPk*	::= **TCP** *tpSrc tpDst payload*
		| **Unknown** *payload*

Figure 4.3: Logical packet structure.

patterns that are closed under the algebraic operations used by our compiler and flow table optimizer.

Roadmap. The rest of this chapter develops techniques for establishing that a given packet-processing function is implemented correctly by an OpenFlow network. More specifically, we tackle the problem of verifying high-level programming abstractions, using NetCore [74] as a concrete instance of a high-level network language. The next section presents NetCore in detail. The following sections describe general and reusable techniques for establishing the correctness of SDN controllers, including NetCore.

4.3 NetCore

This section presents the highest layer of our verified stack: the NetCore language. A NetCore program specifies how the switches process packets at each hop through the network. More formally, a program denotes a total function from port-packet pairs to multisets of port-packet pairs. The syntax and semantics of a core NetCore fragment are shown in Fig. 4.4. To save space, we have elided several header fields and operators not used in this chapter.

We can build a NetCore program that implements the example from the previous section by composing several smaller NetCore program fragments. The first fragment forwards traffic

$$
\begin{array}{lll}
\text{Switch ID} & sw \in \mathbb{N} & \\
\text{Port ID} & pt \in \mathbb{N} & \\
\text{Headers} & h ::= \mathbf{dlSrc} \mid \mathbf{dlDst} & \text{MAC } \textit{address} \\
& \quad \mid \mathbf{dlTyp} & \textit{Ethernet frame type} \\
& \quad \mid \mathbf{nwSrc} \mid \mathbf{nwDst} & \text{IP } \textit{address} \\
& \quad \mid \mathbf{nwProto} & \text{IP } \textit{protocol code} \\
& \quad \mid \mathbf{tpSrc} \mid \mathbf{tpDst} & \textit{transport port} \\
\text{Predicate} & pr ::= \star & \textit{wildcard} \\
& \quad \mid h = n & \textit{match header} \\
& \quad \mid \mathbf{on}\ \ sw & \textit{match switch} \\
& \quad \mid \mathbf{at}\ \ \_ & \textit{match inport} \\
& \quad \mid \mathbf{not}\ pr & \textit{predicate negation} \\
& \quad \mid pr_1\ \mathbf{and}\ pr_2 & \textit{predicate conjunction} \\
\text{Program} & \phi ::= pr \Rightarrow \{\!|pt_1 \cdots pt_n|\!\} & \textit{basic program} \\
& \quad \mid \phi_1 \uplus \phi_2 & \textit{program union} \\
& \quad \mid \mathbf{restrict}\ \phi\ \mathbf{by}\ pr & \textit{program restriction}
\end{array}
$$

$\boxed{[\![pr]\!]\ sw\ pt\ pk}$

$$
\begin{array}{l}
[\![\star]\!]\ sw\ pt\ pk = \mathbf{true} \\
[\![\mathbf{dlSrc}{=}n]\!]\ sw\ pt\ (\mathbf{Eth}\ dlSrc\ \_\ \_\ \_) = dlSrc{=}n \\
[\![\mathbf{nwSrc}{=}n]\!]\ sw\ pt\ (\mathbf{Eth}\ \_\ \_\ \_\ (\mathbf{IP}\ nwSrc\ \_\ \_\ \_)) = nwSrc{=}n \\
[\![\mathbf{nwSrc}{=}n]\!]\ sw\ pt\ (\mathbf{Eth}\ \_\ \_\ \_\ (\mathbf{Unknown}\ \_)) = \mathbf{false} \\
\qquad\qquad \cdots \\
[\![\mathbf{on}\ \ sw']\!]\ sw\ pt\ pk = sw{=}sw' \\
[\![\mathbf{at}\ \ pt']\!]\ sw\ pt\ pk = pt{=}pt' \\
[\![\mathbf{not}\ pr]\!]\ sw\ pt\ pk = \neg([\![pr]\!]\ sw\ pt\ pk) \\
[\![pr_1\ \mathbf{and}\ pr_2]\!]\ sw\ pt\ pk = [\![pr_1]\!]\ sw\ pt\ pk \wedge [\![pr_2]\!]\ sw\ pt\ pk
\end{array}
$$

$\boxed{[\![\phi]\!]\ sw\ pt\ pk = \{\!|(pt_1, pk_1) \cdots (pt_n, pk_n)|\!\}}$

$$
\begin{array}{l}
[\![pr \Rightarrow \{\!|pt_1 \cdots pt_n|\!\}]\!]\ sw\ pt\ pk = \\
\quad \mathbf{if}\ [\![pr]\!]\ sw\ pt\ pk\ \mathbf{then}\ \{\!|(pt_1, pk) \cdots (pt_n, pk)|\!\}\ \mathbf{else}\ \{\!|\,|\!\} \\
[\![\phi_1 \uplus \phi_2]\!]\ sw\ pt\ pk = \\
\quad [\![\phi_1]\!]\ sw\ pt\ pk \uplus [\![\phi_2]\!]\ sw\ pt\ pk \\
[\![\mathbf{restrict}\ \phi\ \mathbf{by}\ pr]\!]\ sw\ pt\ pk = \\
\quad \{\!|(pt', pk') \mid (pt', pk') \in [\![pg]\!]\ sw\ pt\ pk \wedge [\![pr]\!]\ sw\ pt\ pk|\!\}
\end{array}
$$

Figure 4.4: NetCore syntax and semantics (extracts).

to WEB:

$$\phi_1 \triangleq \textbf{at LB and dlDst=WEB} \Rightarrow \{|\mathsf{WEB}|\}$$

This basic program consists of a predicate pr and a multiset of actions $\{|pt_1 \cdots pt_n|\}$. The predicate denotes a set of port-packet pairs, and the actions denote the ports (if any) where those packets should be forwarded on the next hop. In this instance, the predicate denotes the set of all packets whose Ethernet destination (**dlDst**) address has the specified value, and whose inport is LB, and the actions denote a transformation that forwards matching packets to port 1. Note that we represent packets as nested sequences of frames (Ethernet, IP, TCP, etc.) as shown in Fig. 4.3. NetCore provides predicates for matching on well-known header fields as well as logical operators such as **and** and **or**, unlike hardware switches, which only provide prioritized sets of rules.

The next fragment is similar to ϕ_1, but forwards traffic to LB instead of WEB:

$$\phi_2 \triangleq \textbf{at WORLD and dlDst=}\mathit{WEB} \Rightarrow \{|\mathsf{LB}|\}$$

Using the union operator, we can combine these programs into a single program that implements forwarding for HTTP traffic:

$$\phi_{\mathsf{WEB}} \triangleq \phi_1 \uplus \phi_2$$

Semantically, the $\uplus$ operator produces the (multiset) union of the results produced by each sub-program. Using the restriction operator **restrict by** , we can limit this forwarding policy to web traffic:

$$\textbf{restrict } \phi_{\mathsf{WEB}} \textbf{ by tpDst=}22$$

Similarly, we can define the forwarding policy for traffic through the firewall:

$$\phi_1' \triangleq \textbf{at } \mathsf{WORLD} \textbf{ and } \textbf{not dlDst}{=}\mathit{WEB} \Rightarrow \{|\mathsf{FW}|\}$$
$$\phi_2' \triangleq \textbf{at } \mathsf{FW} \textbf{ and } \textbf{not dlDst}{=}\mathit{WEB} \Rightarrow \{|\mathsf{INTRANET}|\}$$
$$\phi_{\mathsf{FW}} \triangleq \phi_1' \uplus \phi_2'$$

Finally, we can add the security policy using the **restrict by** operator, which restricts a program by a predicate:

$$\textbf{restrict } (\phi_{\mathsf{WEB}} \uplus \ \phi_{\mathsf{FW}}) \textbf{ by } (\textbf{not tpDst}{=}22)$$

This program is similar the previous one, but drops SSH traffic.

The advantages of using a declarative language such as NetCore should be clear: it provides abstractions that make it easy to establish network-wide properties through compositional reasoning. For example, simply by inspecting the final program and using the denotational semantics (Fig. 4.4), we can easily verify that the network blocks SSH traffic, forwards HTTP traffic to the middlebox, and other forwards traffic to INTRANET. In particular, even though a controller, switches, flow tables, forwarding rules, are all involved in implementing this program, we do not have to reason about them! This is in contrast to lower-level controller platforms, which require programmers to explicitly construct switch-level forwarding rules, issue messages to install those rules on switches, and reason about the asynchronous interactions between switches and controller. Of course, the complexity of the underlying system is not eliminated, but relocated from the programmer to the language implementers. This is an efficient tradeoff: functionality common to many programs can be implemented just once, proved correct, and reused broadly.

$$
\begin{array}{lll}
\text{Wildcard} & w & ::= n \mid \star \\
\text{Pattern} & pat & ::= \{\mathbf{dlSrc} = w, \mathbf{dlDst} = w, \mathbf{dlTyp} = w, \\
 & & \quad \mathbf{nwSrc} = w, \mathbf{nwDst} = w, \mathbf{nwProto} = w, \\
 & & \quad \mathbf{tpSrc} = w, \mathbf{tpDst} = w\} \\
\text{Flow table} & FT & \in \{\!|n \times pat \times \{\!|pt|\!\}|\!\}
\end{array}
$$

$\boxed{[\![FT]\!]\ pt\ pk \rightsquigarrow \{\!|pt_1 \cdots pt_n|\!\} \times \{\!|pk_1 \cdots pk_m|\!\}}$

$$
\frac{\begin{array}{c}\exists (n, pat, \{\!|pt_1 \cdots pt_n|\!\}) \in FT. \\ pk \# pat = \mathbf{true} \\ \forall (n', pat', pts') \in FT.\ n' > n \Rightarrow \\ pk \# pat' = \mathbf{false}\end{array}}{[\![FT]\!]\ pt\ pk \rightsquigarrow (\{\!|(pt_1) \cdots (pt_n)|\!\}, \{\!|\,|\!\})} \quad \text{(Matched)}
$$

$$
\frac{\forall (n, pat, pts) \in FT \qquad pk \# pat = \mathbf{false}}{[\![FT]\!]\ pt\ pk \rightsquigarrow (\{\!|\,|\!\}, \{\!|(pt, pk)|\!\})} \quad \text{(Unmatched)}
$$

$\boxed{pk \# pat}$

$$
\begin{array}{l}
(\mathbf{Eth}\ dlSrc\ dlDst\ dlTyp\ nwPk) \# pat = \\
\quad dlSrc \sqsubseteq pat.\mathbf{dlSrc}\ \wedge\ dlDst \sqsubseteq pat.\mathbf{dlDst}\ \wedge \\
\quad dlTyp \sqsubseteq pat.\mathbf{dlTyp}\ \wedge \\
\quad (pat.\mathbf{dlTyp} = \texttt{0x800} \Rightarrow nwPk \#_{nw} pat)
\end{array}
$$

$\boxed{nwPk \#_{nw} pat}$

$$
\begin{array}{l}
(\mathbf{IP}\ nwSrc\ nwDst\ nwProto\ tpPk) \#_{nw} pat = \\
\quad nwSrc \sqsubseteq pat.\mathbf{nwSrc} \wedge nwDst \sqsubseteq pat.\mathbf{nwDst}\ \wedge \\
\quad nwProto \sqsubseteq pat.\mathbf{nwProto}\ \wedge \\
\quad (pat.\mathbf{nwProto} = \texttt{6} \Rightarrow tpPk \#_{tp} pat) \\
(\mathbf{Unknown}\ payload) \#_{nw} pat = \mathbf{true}
\end{array}
$$

$\boxed{tpPk \#_{tp} pat}$

$$
\begin{array}{l}
\quad (\mathbf{TCP}\ tpSrc\ tpDst\ payload) \#_{tp} pat = \\
\quad\quad tpSrc \sqsubseteq pat.\mathbf{tpSrc} \wedge tpDst \sqsubseteq pat.\mathbf{tpDst} \\
\quad \mathbf{Unknown}\ payload \#_{tp} pat = \mathbf{true}
\end{array}
$$

$\boxed{n \sqsubseteq w}$

$$
m \sqsubseteq n = m{=}n \qquad n \sqsubseteq \star = \mathbf{true}
$$

Figure 4.5: Flow table syntax and semantics.

4.4 Flow Tables

The first step toward executing a NetCore program in an SDN involves compiling it to a prioritized set of forwarding rules—a *flow table*. Flow tables are an intermediate representation that play a similar role in NetCore to register transfer language (RTL) in traditional compilers. Flow tables are more primitive than NetCore programs because they lack the logical structure induced by NetCore operators such as union, intersection, negation, and restriction. Also, the patterns used to match packets in flow tables are more restrictive than NetCore predicates. And unlike NetCore programs, which denote total functions, flow tables are partial: switches redirect unmatched packets to the controller.

As defined in Fig. 4.5, a *flow table* consists of a multiset of rules (n, pat, pts) where n is an integer priority, *pat* is a pattern, and *pts* is a multiset of ports. A *pattern* is a record that associates each header field to either an integer constant n or the special *wildcard* value $\star$. When writing flow tables, we often elide headers set to $\star$ in patterns as well as priorities when they are clear from context.

Pattern semantics. The semantics of patterns is given by the function $pk\#pat$, as defined in Fig. 4.5. This turns out to be subtly complicated, due to the representation of packets as sequences of nested frames—a pattern contains a (possibly wildcarded) field for every header field, but not all packets contain every header field. Some fields only exist in specific frame types (**dlTyp**) or protocols (**nwProto**). For example, only IP packets (**dlTyp** = `0x800`) have IP source and destination addresses. Likewise, TCP (**nwProto** = `6`) and UDP (**nwProto** = `17`) packets have source and destination ports, but ICMP (**nwProto** = `1`) packets do not.

To match on a given field, a pattern must specify values for all other fields it depends on. For example, to match on IP addresses, the pattern must also specify that the Ethernet frame type is IP:

$$\{\textbf{dlTyp} = \texttt{0x800}, \textbf{nwSrc} = \texttt{10.0.0.1}\}$$

If the frame type is elided, the value of the dependent header is silently ignored and the pattern is equivalent to a wildcard:

$$\{\textbf{nwSrc} = \texttt{10.0.0.1}\} \equiv \{\}$$

In effect, patterns not only match packets, but also determine how they are parsed. This behavior, which was ambiguous in early versions of the OpenFlow specification (and later fixed), has lead to real bugs in existing controllers (Section 4.5). Although unintuitive for programmers, this behavior is completely consistent with how packet processing is implemented in modern switch hardware.

Flow table semantics. The semantics of flow tables is given by the relation $[\![\cdot]\!]$. The relation has two cases: one for matched packets and another for unmatched packets. Each flow table entry is a tuple containing a priority n, pattern pat, and a multiset of ports $\{\!|pt_1 \cdots pt_n|\!\}$. Given a packet and its input port, the semantics forwards the packet to all ports in the multiset associated with the highest-priority matching rule in the table. Otherwise, if no matching rule exists, it diverts the packet to the controller. In the formal semantics, the first component of the result pair represents forwarded packets while the second component represents diverted packets. Note that flow table matching is non-deterministic if there are multiple matching entries with the same priority. This has serious implications for a compiler—*e.g.*, naively combining flow tables with overlapping priorities could produce incorrect results. In the NetCore compiler, we avoid this issue by always working with unambiguous and total flow tables.

$\boxed{\mathcal{P} : sw \times pr \to [(pat, \mathbf{bool})]}$

$$
\begin{array}{lll}
\mathcal{P}(sw, \mathbf{dlSrc} = n) & = & [(\{\mathbf{dlSrc} = n\}, \mathbf{true})] \\
\mathcal{P}(sw, \mathbf{nwSrc} = n) & = & [(\{\mathbf{dlTyp} = \texttt{0x800}, \mathbf{nwSrc} = n\}, \mathbf{true})] \\
 & \cdots & \\
\mathcal{P}(sw, \mathbf{at}\ sw) & = & [(\star, \mathbf{true})] \\
\mathcal{P}(sw, \mathbf{at}\ sw') & = & [(\star, \mathbf{false})] \quad \text{where } sw \neq sw' \\
\mathcal{P}(sw, \mathbf{not}\ pr) & = & [(pat_1, \neg b_1) \cdots (pat_n, \neg b_n), (\star, \mathbf{false})] \\
\quad \text{where } [(pat_1, b_1) \cdots (pat_n, b_n)] = \mathcal{P}(sw, pr) & & \\
\mathcal{P}(sw, pr\ \mathbf{and}\ pr') & = & \\
\quad [(pat_1 \cap pat'_1, b_1 \wedge b'_1) \cdots (pat_m \cap pat'_n, b_m \wedge b'_n)] & & \\
\quad\quad \text{where } [(pat_1, b_1) \cdots (pat_m, b_m)] = \mathcal{P}(sw, pr) & & \\
\quad\quad \text{where } [(pat'_1, b'_1) \cdots (pat'_n, b'_n)] = \mathcal{P}(sw, pr') &
\end{array}
$$

$\boxed{\mathcal{C} : sw \times \phi \to [(pat, pt)]}$

$$
\begin{array}{l}
\mathcal{C}(sw, pr \Rightarrow pt) = [(pat_1, pt_1) \cdots (pat_n, pt_n), (\star, \{\!|\}\!)] \\
\quad \text{where } [(pat_1, b_1), \cdots, (pat_n, b_n)] = \mathcal{P}(sw, pr) \\
\quad \text{where } pt_i = pt \text{ if } b_i = \mathbf{true} \\
\quad \text{where } pt_i = \{\!|\}\!\} \text{ if } b_i = \mathbf{false} \\
\mathcal{C}(sw, \phi \uplus \phi') \quad = \\
\quad [(pat_1 \cap pat'_1, pt_1 \uplus pt'_1), \cdots, (pat_m \cap pat'_n, pt_m \uplus pt'_n)] +\!\!+ \\
\quad [(pat_1, pt_1) \cdots (pat_m, pt_m)] +\!\!+ \\
\quad [(pat'_1, pt'_1) \cdots (pat'_n, pt'_n)] \\
\quad\quad \text{where } [(pat_1, pt_1) \cdots (pat_m, pt_m)] = \mathcal{P}(sw, \phi) \\
\quad\quad \text{where } [(pat'_1, pt'_1) \cdots (pat'_n, pt'_n)] = \mathcal{P}(sw, \phi')
\end{array}
$$

Figure 4.6: NetCore compilation.

4.5 Verified NetCore Compiler

With the syntax and semantics of NetCore and flow tables in place, we now present a verified compiler for NetCore. The compiler takes programs as input and generates a set of flow tables as output, one for every switch. The compilation algorithm is based on previous work [74], but we have verified its implementation in Coq. While building the compiler, we found two serious bugs in the original algorithm related to the handling of (unnatural) patterns in the compiler and flow table optimizer.

The compilation function $\mathcal{C}$, defined in Fig. 4.6, generates a flow table for a given switch sw. It uses the auxiliary function $\mathcal{P}$ to compile predicates. The compiler produces a list of pattern-action pairs, but priority numbers are implicit: the pair at the head has highest priority and each successive pair has lower priority.

Because NetCore programs denote total functions, packets not explicitly matched by any predicate are dropped. In contrast, flow tables divert unmatched packets to the controller. The compiler resolves this discrepancy by adding a catch-all rule that drops unmatched packets. For example, we compile the NetCore policy that forwards packets coming from the MAC address $H1$ to port 5 into a flow table with two rules, one that forwards these packets to port 5, and a lower priority rule that matches all (remaining) packets and drops them:

$$\mathcal{C}(sw, \mathbf{dlSrc} = H1 \Rightarrow \{\!|5|\!\}) = \{\!|(2, \{\mathbf{dlSrc} = H1\}, \{\!|5|\!\}), (1, \star, \{\!|\,|\!\})|\!\}$$

The key operator used by the compiler constructs the cross-product of the flow tables provided as input. This operator can be used to compute intersections and unions of flow tables. Note that implementing union in the obvious way—by concatenating flow tables—would be wrong. The cross-product operator performs an element-wise intersection of the input flow tables and then merges their actions. To compile a union, we first use cross-product to build a flow table that represents the intersection, and then concatenate the flow tables for the sub-policies at lower priority. For example, the following NetCore program,

$$\mathbf{dlSrc} = H1 \Rightarrow \{\!|5|\!\} \uplus \mathbf{dlDst} = H2 \Rightarrow \{\!|10|\!\}$$

compiles to a flow table:

Priority	Pattern	Action
4	$\{\mathbf{dlSrc} = H1, \mathbf{dlDst} = H2\}$	$\{\!|5, 10|\!\}$
3	$\{\mathbf{dlSrc} = H1\}$	$\{\!|5|\!\}$
2	$\{\mathbf{dlDst} = H2\}$	$\{\!|10|\!\}$
1	$\star$	$\{\!|\,|\!\}$

The first rule matches all packets that match both sub-programs, while the second and third rules match packets only matched by the left and the right programs respectively. The final rule drops all other packets. The compilation of other predicates uses similar manipulations on flow tables.

We have built a large library of flow table manipulation operators in Coq, along with several lemmas that state useful algebraic properties about these operators. With this library, proving the correctness theorem for the NetCore compiler is simple—only about 200 lines of code in Coq.

Theorem 7 (NetCore Compiler Soundness)**.** *For all NetCore programs ϕ, switches sw, ports pt, and packets pk we have $[\![\mathcal{C}(sw, \phi)]\!]\ pt\ pk = [\![\phi]\!]\ sw\ pt\ pk$.*

Intuitively, this theorem states that a flow table compiled from a NetCore program for a switch *sw* has the same behavior as the NetCore program evaluated on packets at *sw*.

Compiler bugs. In the course of our work, we discovered that several unverified compilers from high-level network programming languages to flow tables suffer from bugs due to subtle pattern semantics. Section 4.4 described inter-field dependencies in patterns. For example, to match packets from IP address `10.0.0.1`, we write

$$\{\mathbf{nwSrc} = \texttt{10.0.0.1}, \mathbf{dlTyp} = \texttt{0x800}\}$$

and if we omit the **dlTyp** field, the IP address is silently ignored. This unintuitive behavior has led to bugs in PANE [22] and Nettle [103] as well as an unverified version of NetCore [74]. To illustrate, consider the following program:

$$\mathbf{nwSrc} = \texttt{10.0.0.1} \Rightarrow \{|5|\}$$

In NetCore, this program matches all IP packets from `10.0.0.1` and forwards them out port 5. But the original NetCore compiler produced the following flow table for this program:

Priority	Pattern	Action
2	{**nwSrc** = `10.0.0.1`}	{|5|}
1	$\star$	{||}

In OpenFlow, because the first pattern does not specify **dlTyp** = `0x800`, it is actually equivalent to the all-wildcard pattern and this flow table sends *all* traffic out port 5. Both PANE and Nettle have similar bugs. Nettle has a special case to handle patterns with IP addresses that do not specify **dlTyp** = `0x800`, but it does not correctly handle patterns that specify a transport port number but not the **nwProto** field. PANE suffers from the same bug. Even worse, these invalid patterns lead to further bugs when flow tables are combined and optimized by the compiler.

Natural patterns. The verified NetCore compiler does not suffer from the bug above. In our formal development, we require that all patterns manipulated by the compiler be what we call *natural patterns*. A natural pattern has the property that if the pattern specifies the value of a field, then all of that field's dependencies are also met. This rules out patterns such as {**nwSrc** = `10.0.0.1`}, which omits the Ethernet frame type necessary to parse the IP address. Natural patterns are easy to define using dependent types in Coq. Moreover, we can calculate the cross-product of two natural patterns by intersecting fields point-wise. Hence, it is easy to prove that natural patterns are closed under intersection.

Lemma 3. *If pat_1 and pat_2 are natural patterns, then $pat_1 \cap pat_2$ is also a natural pattern.*

Another important property is that all patterns can be expressed as some equivalent natural pattern (where patterns are equivalent if they denote the same set of packets). This property tells us that we do not lose expressiveness by restricting to natural patterns.

Lemma 4. *If pat is an arbitrary pattern, then there exists a natural pattern pat′, such that pat* $\equiv$ *pat′.*

These lemmas are used extensively in the proofs of correctness for our compiler and flow table optimizer.

Flow table optimizer. The basic NetCore compilation algorithm described so far generates flow tables that correctly implement the semantics of the input program. But many flow tables have redundant entries that could be safely removed. For example, a naive compiler might translate the program $(\star \Rightarrow \{|5|\})$ to the flow table $\{|(2, \star, \{|5|\}), (1, \star, \{||\})|\}$, which is equivalent to $\{|(2, \star, \{|5|\})|\}$. Worse, because the compilation rule for union uses a cross-product operator to combine the flow tables computed for sub-programs, the output can be exponentially larger than the input. Without an optimizer, such a naive compiler is essentially useless—*e.g.*, we built an unoptimized implementation of the algorithm in Fig. 4.6 and found that it ran out of memory when compiling a program consisting of just 9 operators.

Our compiler is parameterized on a function $\mathcal{O} : FT \to FT$, that it invokes at each recursive call. Because even simple policies can see a combinatorial explosion during compilation, this inline reduction is necessary. We stipulate that $\mathcal{O}$ must produce equivalent flow tables: $[\![\mathcal{O}(FT)]\!] = [\![FT]\!]$.

We have built an optimizer that eliminates low-priority entries whose patterns are fully subsumed by higher-priority rules and proved that it satisfies the above condition in Coq. Although this optimization is quite simple, it is effective in practice. In addition, earlier attempts to implement this optimization in NetCore had a bug that incorrectly identified certain rules as overlapping which we did not discover until developing this proof. The PANE optimizer also had a bug—it assumed that combining identical actions is always idempotent.

Switch	$S ::= \mathbb{S}(sw, pts, FT, in_p, out_p, in_m, out_m)$		Ports on switch	$pts \in \{pt\}$
Controller	$C ::= \mathbb{C}(\sigma, f_{in}, f_{out})$		Input/output buffers	$in_p, out_p \in \{\!\|(pt, pk)\|\!\}$
Link	$L ::= \mathbb{L}((sw_{src}, pt_{src}), pks, (sw_{dst}, pt_{dst}))$		Messages from controller	$in_m \in \{\!\|SM\|\!\}$
Link to Controller	$M ::= \mathbb{M}(sw, SMS, CMS)$		Messages to controller	$out_m \in \{\!\|CM\|\!\}$
	Devices			**Switch Components**
Controller state	σ		Queue from controller	$SMS \in [SM_1 \cdots SM_n]$
Controller input relation	$f_{in} \in sw \times CM \times \sigma \rightsquigarrow \sigma$		Queue to controller	$CMS \in [CM_1 \cdots CM_n]$
Controller output relation	$f_{out} \in \sigma \rightsquigarrow sw \times SM \times \sigma$			**Controller Link**
	Controller Components			

From controller	$SM ::= \textbf{FlowMod}\ \delta \mid \textbf{PktOut}\ pt\ pk \mid \textbf{BarrierRequest}\ n$
To controller	$CM ::= \textbf{PktIn}\ pt\ pk \mid \textbf{BarrierReply}\ n$
Table update	$\delta ::= \textbf{Add}\ n\ pat\ act \mid \textbf{Del}\ pat$

Figure 4.7: Featherweight OpenFlow syntax

Both of these bugs led to incorrect behavior.

4.6 Featherweight OpenFlow

The next step towards executing NetCore programs is a controller that configures the switches in the network. To prove that such a controller is correct, we need a model of the network. Unfortunately, the OpenFlow 1.0 specification, consisting of 42 pages of informal prose and C definitions, is not amenable to rigorous proof.

This section presents Featherweight OpenFlow, a detailed operational model that captures the essential features of OpenFlow networks, and yet still fits on a single page. The model elides a number of details such as error codes, counters, packet modification, and advanced configuration options such as the ability to enable and disable ports. But it does include all of the features related to how packets are forwarded and how flow tables are modified. Many existing SDN bug-finding and property-checking tools are based on similar

$$\frac{[\![FT]\!](pt, pk) \rightsquigarrow (\{\!|pt'_1 \cdots pt'_n|\!\}, \{\!|pk'_1 \cdots pk'_m|\!\}) \quad out = \{\!|\mathbf{PktIn}\ \ pt\ pk'_1 \cdots \mathbf{PktIn}\ \ pt\ pk'_m|\!\}}{\begin{array}{c}\mathbb{S}(sw, \_, FT, \{\!|(pt, pk)|\!\} \uplus in_p, out_p, \_, out_m) \\ \xrightarrow{(sw,pt,pk)} \quad \mathbb{S}(sw, \_, FT, in_p, \{\!|(pt'_1, pk) \cdots (pt'_n, pk)|\!\} \uplus out_p, \_, out \uplus out_m)\end{array}} \text{(Fwd)}$$

$$\frac{}{\mathbb{S}(sw, \ldots, \{\!|(pt, pk)|\!\} \uplus out_p, \ldots) \mid \mathbb{L}((sw, pt), pks, \_) \longrightarrow \mathbb{S}(sw, \ldots, out_p, \ldots) \mid \mathbb{L}((sw, pt), [pk] +\!\!+ pks, \_)} \text{(Wire-Send)}$$

$$\frac{}{\mathbb{L}(\_, pks +\!\!+ [pk], (sw, pt)) \mid \mathbb{S}(sw, \ldots, in_p, \ldots) \longrightarrow \mathbb{L}(\_, pks, (sw, pt)) \mid \mathbb{S}(sw, \ldots, \{\!|(pt, pk)|\!\} \uplus in_p, \ldots)} \text{(Wire-Recv)}$$

$$\frac{}{\mathbb{S}(\ldots, FT, \ldots, \{\!|\mathbf{FlowMod\ Add}\ m\ pat\ act|\!\} \uplus in_m, \_) \longrightarrow \mathbb{S}(\ldots, FT \uplus \{\!|(m, pat, act)|\!\}, \ldots, in_m, \_)} \text{(Add)}$$

$$\frac{FT_{rem} = \{\!|(n', pat', act') \mid (n', pat', act') \in FT \text{ and } pat \neq pat'|\!\}}{\mathbb{S}(\ldots, FT, \ldots, \{\!|\mathbf{FlowMod\ Del}\ pat|\!\} \uplus in_m, \_) \longrightarrow \mathbb{S}(\ldots, FT_{rem}, \ldots, in_m, \_)} \text{(Del)}$$

$$\frac{pt \in pts}{\mathbb{S}(\_, pts, \ldots, out_p, \{\!|\mathbf{PktOut}\ pt\ pk|\!\} \uplus in_m, \_) \longrightarrow \mathbb{S}(\_, pts, \ldots, \{\!|(pt, pk)|\!\} \uplus out_p, in_m, \_)} \text{(PktOut)}$$

$$\frac{f_{out}(\sigma) \rightsquigarrow (sw, SM, \sigma')}{\mathbb{C}(\sigma, \_, \_) \mid \mathbb{M}(sw, SMS, \_) \longrightarrow \mathbb{C}(\sigma', \_, \_) \mid \mathbb{M}(sw, [SM] +\!\!+ SMS, \_)} \text{(Ctrl-Send)}$$

$$\frac{f_{in}(sw, \sigma, CM) \rightsquigarrow \sigma'}{\mathbb{C}(\sigma, f_{in}, \_) \mid \mathbb{M}(sw, \_, CMS +\!\!+ [CM]) \longrightarrow \mathbb{C}(\sigma', f_{in}, \_) \mid \mathbb{M}(sw, \_, CMS)} \text{(Ctrl-Recv)}$$

$$\frac{SM \neq \mathbf{BarrierRequest}\ n}{\mathbb{M}(sw, SMS +\!\!+ [SM], \_) \mid \mathbb{S}(sw, \ldots, in_m, \_) \longrightarrow \mathbb{M}(sw, SMS, \_) \mid \mathbb{S}(sw, \ldots, \{\!|SM|\!\} \uplus in_m, \_)} \text{(Switch-Recv-Ctrl)}$$

$$\frac{}{\begin{array}{c}\mathbb{M}(sw, SMS +\!\!+ [\mathbf{BarrierRequest}\ n], \_) \mid \mathbb{S}(sw, \ldots, \{\!|\}\!\}, out_m) \\ \longrightarrow \quad \mathbb{M}(sw, SMS, \_) \mid \mathbb{S}(sw, \ldots, \{\!|\}\!\}, \{\!|\mathbf{BarrierReply}\ n|\!\} \uplus out_m)\end{array}} \text{(Switch-Recv-Barrier)}$$

$$\frac{}{\mathbb{S}(sw, \ldots, \{\!|CM|\!\} \uplus out_m) \mid \mathbb{M}(sw, \_, CMS) \longrightarrow \mathbb{S}(sw, \ldots, out_m) \mid \mathbb{M}(sw, \_, [CM] +\!\!+ CMS)} \text{(Switch-Send-Ctrl)}$$

$$\frac{Sys_1 \longrightarrow Sys'_1}{Sys_1 \mid Sys_2 \longrightarrow Sys'_1 \mid Sys_2} \text{(Congruence)}$$

Figure 4.8: Featherweight OpenFlow semantics.

(informal) models [52, 49, 14]. We believe Featherweight OpenFlow could also serve as a foundation for these tools.

4.6.1 OpenFlow Semantics

Initially, every switch has an empty flow table that diverts all packets to the controller. Using **FlowMod** messages, the controller can insert new table entries to have the switch process packets itself. A non-trivial program may compile to several thousand flow table entries, but

FlowMod messages only add a single entry at a time. In general, many **FlowMod** messages will be needed to fully configure a switch. However, OpenFlow is designed to give switches a lot of latitude to enable efficient processing, often at the expense of programmability and understandability:

- **Pattern semantics.** As discussed in preceding sections, the semantics of flow tables is non-trivial: patterns have implicit dependencies and flow tables can have multiple, overlapping entries. (The OpenFlow specification itself notes that scanning the table to find overlaps is expensive.) Therefore, it is up to the controller to avoid overlaps that introduce non-determinism.
- **Packet reordering.** Switches may reorder packets arbitrarily. For example, switches often have both a "fast path" that uses custom packet-processing hardware and a "slow path" that processes packets using a slower general-purpose CPU.
- **No acknowledgments.** Switches do not acknowledge when **FlowMod** messages are processed, except when errors occur. The controller can explicitly request acknowledgments by sending a *barrier request* after a **FlowMod**. When the switch has processed the **FlowMod** (and all other messages received before the *barrier request*), it responds with a *barrier reply*.
- **Control message reordering.** Switches may process control messages, including **FlowMod** messages, in any order. This is based on the architecture of switches, where the logical flow table is implemented by multiple physical tables working in parallel—each physical table typically only matches headers for one protocol. To process a rule with a pattern such as $\{\textbf{nwSrc} = \texttt{10.0.0.1}, \textbf{dlTyp} = \texttt{0x800}\}$, which matches headers across several protocols, several physical tables may need to be reconfigured, which takes longer to process than a simple pattern such as $\{\textbf{dlDst} = \texttt{H2}\}$.

Figure 4.8 defines the syntax and semantics of Featherweight OpenFlow, which faithfully models all of these behaviors. The rest of this section discusses the key elements of the model in detail.

4.6.2 Network Elements

Featherweight OpenFlow has four kinds of elements: switches, controllers, links between switches (carrying data packets), and links between switches and the controller (carrying OpenFlow messages). The semantics is specified using a small-step relation, with elements interacting by passing messages and updating their state non-deterministically.

Switches. A switch $\mathbb{S}$ comprises a unique identifier sw, a set of ports pts, and input and output packet buffers in_p and out_p. The buffers are multisets of packets tagged with ports, (pt, pk). In the input buffer, packets are tagged with the port on which they were received. In the output buffer, packets are tagged with the port on which they will be sent out. Since buffers are unordered, switches can process packets in any order. Switches also have a flow table, FT, which determines how the switch processes packets. As detailed in Section 4.4, the table is a collection of flow table entries, where each entry has a priority, pattern and, a multiset of output ports. Each switch also has a multiset of messages to and from the controller, out_m and in_m. There are three kinds of messages from the controller:

- **PktOut** *pt pk* instructs the switch to emit packet *pk* on port *pt*.
- **FlowMod** δ instructs the switch to add or delete entries from its flow table. When δ is **Add** *n pat act*, a new entry is created, whereas **Del** *pat* deletes all entries that match *pat* exactly. In our model, we assume that flow tables on switches can be arbitrarily

large. This is not the case for hardware switches, where the size of flow tables is often constrained by the amount of silicon used, and varies from switch-to-switch. It would be straightforward to modify our model to bound the size of the table on each switch.

- **BarrierRequest** n forces the switch to process all outstanding messages before replying with a **BarrierReply** n message.

Controllers. A controller $\mathbb{C}$ is defined by its local state σ, an input relation f_{in}, and an output relation f_{out}. The local state and these relations are application-specific, so Featherweight OpenFlow can be instantiated with any controller whose behavior can be modeled in this way. The f_{out} relation sends a message to a switch while f_{in} receives a message from a switch. Both relations update the state σ. There are two kinds of messages a switch can send to the controller:

- **PktIn** *pt pk* indicates that packet *pk* was received on *pt* and did not match any entry in the flow table.
- **BarrierReply** n indicates that *sw* has processed all messages up to and including a **BarrierRequest** n sent earlier.

Data links. A data link $\mathbb{L}$ is a unidirectional queue of packets between two switch ports. To model bidirectional links we use symmetric unidirectional links. Featherweight OpenFlow does not model packet-loss in links and packet-buffers. It would be easy to extend our model so that packets are lost, for example, with some probability. Without packet loss, a packet traces paths from its source to its destinations (or loops forever). With packet loss, a packet traces a prefix of the complete path given by our model under ideal conditions.

Location	$loc ::= sw \times pt$
Located packet	$lp \ ::= loc \times pk$
Topology	$T \ \in \ loc \rightharpoonup loc$

$\boxed{\phi, T \vdash \{\!|lp|\!\} \overset{lp}{\Rightarrow} \{\!|lp|\!\}}$

$$\frac{lps' = \{\!|(T(sw, pt_{out}), pk) \mid (pt_{out}, pk) \in [\![\phi]\!] \ sw \ pt \ pk|\!\}}{\begin{array}{rl} \phi, T \vdash & \{\!|((sw, pt), pk)|\!\} \uplus \{\!|lp_1 \cdots lp_n|\!\} \overset{(sw,pt,pk)}{\Longrightarrow} \\ & lps' \uplus \{\!|lp_1 \cdots lp_n|\!\} \end{array}}$$

Figure 4.9: Network semantics.

Control links. A control link $\mathbb{M}$ is a bidirectional link between the switch and the controller that contains a queue of controller messages for the switch and a queue of switch messages headed to the controller. Messages between the controller and the switch are sent and delivered in order, but may be processed in any order.

4.7 Verified Run-Time System

So far, we have developed a semantics for NetCore (Section 4.3), a compiler from NetCore to flow tables (Section 4.4), and a low-level semantics for OpenFlow (Section 4.6). To actually execute NetCore programs, we also need to develop a run-time system that installs rules on switches and prove it correct.

4.7.1 NetCore Run-Time System

There are many ways to build a controller that implements a NetCore run-time system. A trivial solution is to simply process all packets on the controller. The controller receives input packets as **PktIn** messages, evaluates them using the NetCore semantics, and emits the outputs using **PktOut** messages.

Of course, we can do much better by using the NetCore compiler to actually generate flow tables and install those rules on switches using **FlowMod** messages. For example, given the following program,

$$\textbf{dlDst} = \texttt{H1}\ \textbf{and not}(\textbf{dlTyp} = \texttt{0x800}) \Rightarrow \{|1|\}$$

the compiler might generate the following flow table,

Priority	Pattern	Action
5	$\{\textbf{dlDst} = \texttt{H1}, \textbf{dlTyp} = \texttt{0x800}\}$	$\{\|\|\}$
4	$\{\textbf{dlDst} = \texttt{H1}\}$	$\{\|1\|\}$
3	$\star$	$\{\|\|\}$

and the controller would emit three **FlowMod** messages:

$$\textbf{Add}\ 5\ \{\textbf{dlDst} = \texttt{H1}, \textbf{dlTyp} = \texttt{0x800}\}\ \{|\|\}$$
$$\textbf{Add}\ 4\ \{\textbf{dlDst} = \texttt{H1}\}\ \{|1|\}$$
$$\textbf{Add}\ 3\ \star\ \{|\|\}$$

However, it would be unsafe to emit just these messages. As discussed in Section 4.6, switches can reorder messages to maximize throughput. This can lead to transient bugs by creating intermediate flow tables that are inconsistent with the intended policy. For example, if the **Add** 3 $\star$ $\{|\|\}$ message is processed first, all packets will be dropped. Alternatively, if **Add** 4 $\{\textbf{dlDst} = \texttt{H1}\}$ $\{|1|\}$ is processed first, traffic that should be dropped will be incorrectly forwarded. Of the six possible permutations, only one has the property that all intermediate states either (i) process packets according to the program, or (ii) send packets to the controller (which can evaluate them using the program). Therefore, to ensure the switch processes the messages in order, the run-time system must intersperse **BarrierRequest** messages between **FlowMod** messages.

Network semantics. The semantics of NetCore presented in Section 4.3 defines how a program processes a single packet at a single switch at a time. But Featherweight OpenFlow models the behavior of an entire network of switches with multiple packets in-flight. To reconcile the difference between these two, we need a *network semantics* that models the processing of all packets in the network. In this semantics (Fig. 4.9), the system state is a multiset of in-flight located packets $\{\!|\, lp \,|\!\}$. At each step, the system:

1. Removes a located packet $((sw, pt), pk)$ from its state,
2. Processes the packet according to the program to produce a new multiset of located packets,
$$\{\!|\, lp_1 \cdots lp_n \,|\!\} = [\![\phi]\!]\ sw\ pt\ pk,$$
3. Transfers these packets to input ports, using the topology, $T(lp_1) \cdots T(lp_n)$, and
4. Adds the transferred packets to the system state.

Note that this approach to constructing a network semantics is not specific to NetCore: any hop-by-hop packet processing function could be used. Below, we refer to any semantics constructed in this way as a *network semantics*.

4.7.2 Run-Time System Correctness

Now we are ready to prove the correctness of the NetCore run-time system. However, rather than proving this directly, we instead develop a general framework for establishing controller correctness, and obtain the result for NetCore as a special case.

Bisimulation equivalence. The inputs to our framework are: (i) the high-level, hop-by-hop function the network is intended to implement, and (ii) the controller implementation, which is required to satisfy natural safety and liveness conditions. Given these parameters, we construct a *weak bisimulation* between the network semantics of the high-level function and an OpenFlow network instantiated with the controller implementation. This construction handles a number of low-level details once and for all, freeing developers to focus on essential controller correctness properties.

We prove a weak (rather than strong) bisimulation[2] because Featherweight OpenFlow models the mechanics of packet processing in much greater detail than in the network semantics. For example, consider a NetCore program that forwards a packet *pk* from one switch to another, say *S1* to *S2*, in a single step. An equivalent Featherweight OpenFlow implementation would require at least three steps: (i) process *pk* at *S1*, move *pk* from the input buffer to the output buffer, (ii) move *pk* from *S1*'s output buffer to the link to *S2*, and (iii) move *pk* from the link to *S2*'s input buffer. If there were other packets on the link (which is likely!), additional steps would be needed. Moreover, *pk* could take an even more circuitous route if it is redirected to the controller.

The weak bisimulation states that the NetCore and Featherweight OpenFlow are indistinguishable modulo internal steps. Hence, any reasoning about the trajectory of a packet at the NetCore level will be preserved in Featherweight OpenFlow.

Observations. To define a weak bisimulation, we need a notion of observation (called an action in the π-calculus). We say that the NetCore network semantics observes a packet

[2]A weak bisimulation identifies states of two systems as equivalent modulo unobservable (internal) transitions. A strong bisimulation does not allow a system to perform unobservable transitions to catch up. For example, a pipelined processor may split one logical operation into two steps, with an unobservable transition between the processing of the steps. The pipelined processor would be weakly bisimilar to a processor that performed the operation in one step, but not strongly bisimilar.

(sw, pt, pk) when it selects the packet from its state—i.e., just before evaluating it. Likewise, a Featherweight OpenFlow program observes a packet (sw, pt, pk) when it removes (pt, pk) from the input buffer on sw to process it using the FWD rule.

Bisimulation relation. Establishing a weak bisimulation requires exhibiting a relation $\approx_{OF}$ between the concrete and abstract states with certain properties. We relate packets located in links and buffers in Featherweight OpenFlow to packets in the abstract network semantics. We elide the full definition of the relation, but describe some of its key characteristics:

- Packets (pt, pk) in input buffers in_p on sw are related to packets $((sw, pt), pk)$ in the abstract state.
- Packets (pt, pk) in output buffers out_p on sw are related to packets located at the other side of the link connected to pt.
- Likewise, packets on a data link (or contained in **PktOut** messages) are related to packets located at the other side of the data link (or the link connected to the port in the message).

Intuitively, packets in output buffers have already been processed and observed. The network semantics moves packets to new locations in one step whereas OpenFlow requires several more steps, but we must not be able to observe these intermediate steps. Therefore, after Featherweight OpenFlow observes a concrete packet pk (in the FWD rule), subsequent copies of pk must be related to packets at the ultimate destination.

The structure of the relation is largely straightforward and dictated by the nature of Featherweight OpenFlow. However, a few parts are application specific. In particular,

packets at the controller and packets sent to the controller in **PktIn** messages may relate to the state in the network semantics in application-specific ways.

Abstract semantics. So far, we have focused on NetCore to build intuitions. But our bisimulation can be obtained for any controller that implements a high-level packet-processing function. We now make this precise with a few additional definitions.

Definition 2 (Abstract Semantics)**.** *An abstract semantics is defined by the following components:*

1. *An abstract packet-processing function on located packets:*

$$f(lp) = \{\!|\, lp_1 \cdots lp_n \,|\!\}$$

2. *An abstraction function,* $c : \sigma \to \{\!|\, lp \,|\!\}$*, that identifies the packets the controller has received but not yet processed.*

Note that the type of the NetCore semantics (Fig. 4.9) matches the type of the function above. In addition, because the NetCore controller simply holds the multiset of **PktIn** messages, the abstraction function is trivial. Given such an abstract semantics, we can lift it to a network semantics $\overset{lp}{\Rightarrow}$ as we did for NetCore.

We say that an abstract semantics is *compatible* with a concrete controller implementation, consisting of a type of controller state σ, and input and output relations f_{in} and f_{out}, if the two satisfy the following conditions relating their behavior:

Definition 3 (Compatibility)**.** *An abstract semantics and controller implementation are* compatible *if:*

1. *The controller ensures that all times packets are either (i) processed by switches in accordance with the packet-processing function or (ii) sent to the controller for processing;*

2. *Whenever the controller receives a packet,*

$$(sw, \textbf{\textit{PktIn}}\ \ pt\ pk, \sigma) \rightsquigarrow \sigma'$$

it applies the packet-processing function f to pk to get a multiset of located packets and adds them to its state

$$c(\sigma') = c(\sigma) \uplus f(pk)$$

3. *Whenever the controller emits a packet,*

$$\sigma \rightsquigarrow (sw, \textbf{\textit{PktOut}}\ \ pt\ pk, \sigma')$$

it removes the packet from its state:

$$c(\sigma') = c(\sigma) \setminus \{\!|(sw, pt, pk)|\!\}$$

4. *The controller eventually processes all packets (sw, pt, pk) in its state $c(\sigma)$ according to the packet-processing function, and*

5. *The controller eventually processes all OpenFlow messages.*

The first property is essential. If it did not hold, switches could process packets contrary to the intended packet-processing relation. Proving it requires reasoning about the messages sent to the switches by the controller. In particular, because switches may reorder messages, barriers must be interspersed appropriately. The second and third properties relate the abstraction function c and the controller implementation. The fourth property requires the controller to correctly process every packet it receives. The fifth property is a liveness condition requiring the controller to eventually process every OpenFlow message. This holds in the absence of failures on the control link and the controller itself.

Given such a semantics, we show that our relation between abstract and Featherweight OpenFlow states and its inverse are weak simulations. This implies that the relation is a weak bisimulation, and thus that the two systems are weakly bisimilar.

Theorem 8 (Weak Bisimulation)**.** *For all compatible abstract semantics and controller implementations, all Featherweight OpenFlow states s and s', and all abstract states t and t':*

- *If $s \approx_{OF} t$ and $s \xrightarrow{(sw,pt,pk)} s'$, then there exists an abstract network state t'' such that $t \xRightarrow{(sw,pt,pk)} t''$ and $s' \approx_{OF} t''$, and*
- *If $s \approx_{OF} t$ and $t \xRightarrow{(sw,pt,pk)} t'$, then there exists a Featherweight OpenFlow state s'', and abstract network states s_i, s_i' such that*

$$s \longrightarrow^* s_i \xrightarrow{(sw,pt,pk)} s_i' \longrightarrow^* s''$$

and $s'' \approx_{OF} t'$.

In this theorem, portions of the $\approx_{OF}$ relation are defined in terms of the controller abstraction function, c supplied as a parameter. In addition, the proofs themselves rely on compatibility (Definition 3).

Finally, we instantiate this theorem for the NetCore controller:

Corollary 2 (NetCore Run-Time Correctness)**.** *The network semantics of NetCore is weakly bisimilar to the concrete semantics of the NetCore controller in Featherweight OpenFlow.*

4.8 Implementation and Evaluation

We have built a complete working implementation of the system described in this chapter, including machine-checked proofs of each of the lemmas and theorems. Our implementation

is available under an open-source license at the following URL:

`http://frenetic-lang.org`

Our system consists of 12 KLOC of Coq, which we extract to OCaml and link against two unverified components:

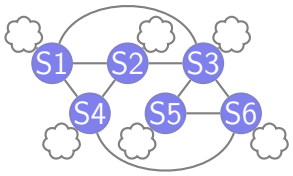
- A library to serialize OpenFlow data types to the OpenFlow wire format. This code is a lightly modified version of the Mirage OpenFlow library [63] (1.4K LOC).
- A module to translate between the full OpenFlow protocol and the fragment used in Featherweight OpenFlow (200 LOC).

We have deployed our NetCore controllers on real hardware and used them to build a number of useful network applications including host discovery, shortest-path routing, spanning tree, access control, and traffic monitoring. Using the union operator, it is easy to compose these modules with others to form larger applications.

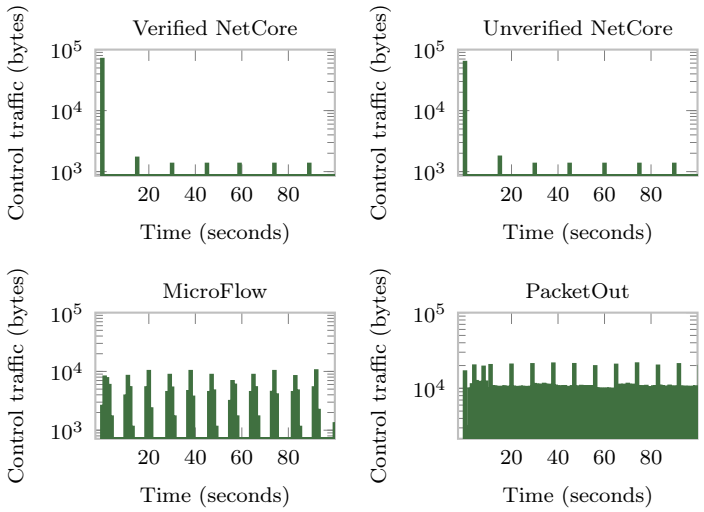
Controller throughput. Controller throughput is important for the performance of SDNs. The CBench [98] tool quantifies controller throughput by flooding the controller with **PktIn** messages and measuring the time taken to receive **PktOut** messages in response. This is a somewhat crude metric, but it is still effective, since any controller must respond to **PktIn** messages. We used CBench to compare the throughput of our verified controller with our previous unverified NetCore controller, written in Haskell, and with the popular POX and NOX controllers, written in Python and C++ respectively. To ensure that the experiment tested throughput and not the application running on it, we had each controller execute a trivial program that floods all packets. We ran the experiment on a dual-core 3.3 GHz Intel i3 with 8GB RAM with Ubuntu 12.04 and obtained the results shown in Fig. 4.10 (a).

Controller	Msgs/sec
Unverified NetCore	26,022
NOX	16,997
Verified NetCore	**9,437**
POX	6,150

(a)



(b)



(c)

Figure 4.10: Experiments: (a) controller throughput results; (b) control traffic topology; (c) control traffic results.

Our unverified NetCore controller is significantly faster than our verified controller. We attribute this to (i) a more mature backend that uses an optimized library from Nettle [103] to serialize messages, and (ii) Haskell's superior multicore support, which the controller exploits heavily. However, despite being slower than the original NetCore, the new controller is still fast enough to be useful—indeed, it is faster than the popular POX controller (although POX is not tuned for performance). We plan to optimize our controller to improve its performance in the future.

Control traffic. Another key factor that affects SDN performance is the amount of traffic that the controller must handle. This metric measures the effectiveness of the controller at compiling, optimizing, and installing forwarding rules rather than processing packets itself. To properly assess a controller on these points, we need a more substantial application than "flood all packets." Using NetCore, we built an application that computes shortest path forwarding rules as well as a spanning tree for broadcast. We ran this program on the six-

switch Waxman topology shown in Fig. 4.10 (b), with two hosts connected to each switch.

In the experiment, every host sent 10 ICMP (ping) broadcast packets along the spanning tree, and received the replies from other hosts along shortest path routes. We used Mininet [37] to simulate the network and collected traffic traces using `tcpdump`. The total amount of network traffic during the experiment was 372 Kb.

We compared our Verified NetCore controller to several others: a (verified) "PacketOut" controller that never installs forwarding rules and processes all packets itself; our previous "Unverified NetCore" controller, written in Haskell; and a reactive "MicroFlow" controller [24] written in Haskell. The results of the experiment are shown in Fig. 4.10 (c). The graphs plot time-series data for every controller, showing the amount of control traffic in each one-second interval. Note that the y axis is on a logarithmic scale.

In the plot for our Verified NetCore controller, there is a large spike in control traffic at the start of the experiment, where the controller sends messages to install the forwarding rules generated from the program. Additional control traffic appears every 15 seconds; these messages implement a simple keep-alive protocol between the controller and switches. The Unverified NetCore controller uses the same compilation and run-time system algorithms as our verified controller, so its plot is nearly identical. The MicroFlow controller installs individual fine-grained rules in response to individual traffic flows rather than proactively compiling complete flow tables. Accordingly, its plot shows that there is much more control traffic than for the two NetCore controllers. The graph shows how traffic spikes when multiple hosts respond simultaneously to an ICMP broadcast. The fourth plot shows the behavior of the PacketOut controller. Because this controller does not install any forwarding rules on the switches, all data traffic flows to the controller and then back into the network.

Although these results are preliminary, we believe they demonstrate that the performance

of our verified NetCore controller can be competitive with other controllers. In particular, our verified controller generates the same flow tables and handles a similar amount of traffic as the earlier unverified NetCore controller, which was written in Haskell. Moreover, our system is not tuned for performance. As we optimize and extend our system, we expect that its performance will only improve.

4.9 Conclusions

This chapter presented a formal foundation for network reasoning: a detailed model of OpenFlow, formalized in the Coq proof assistant, and a machine-verified compiler and run-time system for the NetCore programming language. The main result is a general framework for establishing controller correctness that reduces the proof obligation to a small number of safety and liveness properties.

CHAPTER 5

REASONING ABOUT NETWORK UPDATES

"Nothing endures but change."

—Heraclitus

In this chapter, we show how to move a network between different configurations in such a way that the network preserves the behavior of the configurations, even while it is in transition. We present network update abstractions, implementations and optimizations of those abstractions, and a formal model of updates in software-defined networks to formally specify our abstractions and prove them correct.

The work in this chapter is based upon a 2012 SIGCOMM paper [87] written with Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker.

5.1 Introduction

The techniques in the previous chapters show how to take a network specification and a network program, and build a system that provably satisfies the specification. But real-world networks exist in a constant state of flux. Operators frequently modify routing tables and change access control lists to perform tasks from planned maintenance, to traffic engineering, to patching security vulnerabilities, to migrating virtual machines in a datacenter. Simply implementing a single network program correctly is not sufficient: we need to update the network program over time in response to changes in the network, and still maintain invariants, even while updates are in progress.

But network updates are difficult to perform correctly: even when planned well in advance

Example Application	Policy Change	Desired Property	Practical Implications
Stateless firewall	Changing access control list	No security holes	Admitting malicious traffic
Planned maintenance [28, 85]	Shut down a node/link	No loops/blackholes	Packet/bandwidth loss
Traffic engineering [27, 85]	Changing a link weight	No loops/blackholes	Packet/bandwidth loss
VM migration [19]	Move server to new location	No loops/blackholes	Packet/bandwidth loss
IGP migration [102]	Adding route summarization	No loops/blackholes	Packet/bandwidth loss
Traffic monitoring	Changing traffic partitions	Consistent counts	Inaccurate measurements
Server load balancing [39, 105]	Changing load distribution	Connection affinity	Broken connections
NAT or stateful firewall	Adding/replacing equipment	Connection affinity	Outages, broken connections

Table 5.1: Example changes to network configuration, and the desired update properties.

they can result in disruptions such as transient outages, lost server connections, unexpected security vulnerabilities, hiccups in VoIP calls, or the death of a player's favorite character in an online game.

To address these problems, researchers have proposed a number of extensions to protocols and operational practices that aim to prevent transient anomalies [29, 28, 45, 85, 102]. However, each of these solutions is limited to a specific protocol (*e.g.*, OSPF and BGP) and a specific set of properties (*e.g.*, freedom from loops and blackholes) and increases the complexity of the system considerably. Hence, in practice, network operators have little help when designing a new protocol or trying to ensure an additional property not covered by existing techniques. A list of example applications and their properties is summarized in Table 5.1.

Instead of relying on point solutions for network updates, this chapter presents foundational principles for designing solutions that are applicable to a wide range of protocols and properties. These solutions come with two parts: (1) an abstract interface that offers strong, precise, and intuitive semantic guarantees, and (2) concrete mechanisms that faithfully implement the semantics specified in the abstract interface. Programmers can use the interface to build robust applications on top of a reliable foundation. The mechanisms, while possibly complex, should be implemented once by experts, tuned and optimized, and used

over and over, much like register allocation or garbage collection in a high-level programming language.

Instead of requiring SDN programmers to implement configuration changes using today's low-level interfaces, our high-level, abstract operations allow the programmer to update the configuration of the entire network in one fell swoop. The libraries implementing these abstractions provide strong semantic guarantees about the observable effects of the global updates, and handle all of the details of transitioning between old and new configurations efficiently.

Abstractions Our central abstraction is *per-packet consistency*, the guarantee that every packet traversing the network is processed by exactly one consistent global network configuration. When a network update occurs, this guarantee persists: each packet is processed either using the configuration in place prior to the update, or the configuration in place after the update, but never a mixture of the two. Note that this consistency abstraction is *more* powerful than an "atomic" update mechanism that simultaneously updates all switches in the network. Such a simultaneous update could easily catch many packets in flight in the middle of the network, and such packets may wind up traversing a mixture of configurations, causing them to be dropped or sent to the wrong destination. We also introduce *per-flow consistency*, a generalization of per-packet consistency that guarantees all packets in the same flow are processed with the same configuration. This stronger guarantee is needed in applications such as HTTP load balancers, which need to ensure that all packets in the same TCP connection reach the same server replica to avoid breaking connections.

To support these abstractions, we develop several *update mechanisms* that use features commonly available on OpenFlow switches. Our most general mechanism, which enables

transition between any two configurations, performs a two-phase update of the rules in the new configuration onto the switches. The other mechanisms are optimizations that achieve better performance under circumstances that arise often in practice. These optimizations transition to new configurations in less time, update fewer switches, or fewer rules.

To analyze our abstractions and mechanisms, we develop a simple, formal model that captures the essential features of OpenFlow networks. This model allows us to define a class of network properties, called *trace properties*, that characterize the paths individual packets take through the network. The model also allows us to prove a remarkable result: if *any* trace property P holds of a network configuration prior to a per-packet consistent update as well as after the update, then P also holds continuously throughout the update process. This illustrates the true power of our abstractions: programmers do not need to specify *which* trace properties our system must maintain during an update, because a per-packet consistent update preserves *all* of them! For example, if the old and new configurations are free from forwarding loops, then the network will be loop-free before, during, and after the update. In addition to the proof sketch included in this chapter, this result has been formally verified in the Coq proof assistant [9].

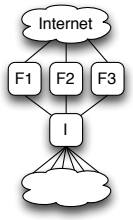
An important and useful corollary of these observations is that it is possible to take any verification tool that checks trace properties of *static* network configurations and transform it into a tool that checks invariance of trace properties as the network configurations evolve *dynamically*—it suffices to check the static policies before and after the update. Indeed, the techniques and systems in the previous chapter all verify trace properties of static configurations, dovetailing perfectly with the developments here.

Contributions This chapter makes the following contributions:

- **Update abstractions:** We propose per-packet and per-flow consistency as canonical, general abstractions for specifying network updates (Sections 5.2 and 5.6).
- **Update mechanisms:** We describe OpenFlow-compatible implementation mechanisms and several optimizations tailored to common scenarios (Sections 5.5 and 5.8).
- **Theoretical model:** We develop a mathematical model that captures the essential behavior of SDNs, and we prove that the mechanisms correctly implement the abstractions (Section 5.3). We have formalized the model and proved the main theorems in the Coq proof assistant.
- **Implementation:** We describe a prototype implementation on top of the OpenFlow/NOX platform (Section 5.8).
- **Experiments:** We present results from experiments run on small, but canonical applications that compare the total number of control messages and rule overhead needed to implement updates in each of these applications (Section 5.8).

5.2 Example

To illustrate the challenges surrounding network updates, consider an example network with one ingress switch I and three "filtering" switches FW1, FW2, and FW3, each sitting between I and the rest of the Internet, as shown on the left side of Figure 5.1. Several classes of traffic are connected to I: untrustworthy packets from Unknown and Guest hosts, and trustworthy packets from Student and Faculty hosts. At all times, the network should enforce a security policy that denies SSH traffic from untrustworthy hosts, but allows all other traffic to pass through the network unmodified. We assume that any of the filtering switches have the capability to perform the requisite monitoring, blocking, and forwarding.



Configuration I

	Type	Action
I	U, G	Forward FW1
	S	Forward FW2
	F	Forward FW3
FW1	*SSH*	Monitor
	$*$	Allow
FW2	$*$	Allow
FW3	$*$	Allow

→

Configuration II

	Type	Action
I	U	Forward FW1
	G	Forward FW2
	S, F	Forward FW3
FW1	*SSH*	Monitor
	$*$	Allow
FW2	*SSH*	Monitor
	$*$	Allow
FW3	$*$	Allow

Figure 5.1: Access control example.

There are several ways to implement this policy, and depending on the traffic load, one may be better than another. Suppose that initially we configure the switches as shown in the leftmost table in Figure 5.1: switch I sends traffic from U and G hosts to FW1, from S hosts to FW2, and from F hosts to FW3. Switch FW1 monitors (and denies) SSH packets and allows all other packets to pass through, while FW2 and FW3 simply let all packets pass through.

Now, suppose the load shifts, and we need more resources to monitor the untrustworthy traffic. We might reconfigure the network as shown in the table on the right of Figure 5.1, where the task of monitoring traffic from untrustworthy hosts is divided between FW1 and FW2, and all traffic from trustworthy hosts is forwarded to FW3. Because we cannot update the network all at once, the individual switches need to be reconfigured one-by-one. However, if we are not careful, making incremental updates to the individual switches can lead to intermediate configurations that violate the intended security policy. For instance, if we start by updating FW2 to deny SSH traffic, we interfere with traffic sent by trustworthy hosts. If, on the other hand, we start by updating switch I to forward traffic according to the new configuration (sending U traffic to FW1, G traffic to FW2, and S and F traffic to FW3), then SSH packets from untrustworthy hosts will incorrectly be allowed to pass through the network. There is one valid transition plan:

1. Update I to forward S traffic to FW3, while continuing to forward U and G traffic to FW1 and F traffic to FW3.
2. Wait until in-flight packets have been processed by FW2.
3. Update FW2 to deny SSH packets.
4. Update I to forward G traffic to FW2, while continuing to forward U traffic to FW1 and S and F traffic to FW3.

But finding this ordering and verifying that it behaves correctly requires performing intricate reasoning about a sequence of intermediate configurations—something that is tedious and error-prone, even for this simple example. Even worse, in some examples it is impossible to find an ordering that implements the transition simply by adding one part of the new configuration at a time (*e.g.*, if we swap the roles of FW1 and FW3 while enforcing the intended security policy). In general, more powerful update mechanisms are needed.

Any energy the programmer devotes to navigating this space would be better spent in other ways. The tedious job of finding a safe sequence of commands that implement an update should be factored out, optimized, and reused across many applications. This is the main achievement of this chapter. To implement the update using our abstractions, the programmer would simply write:

```
per_packet_update(config2)
```

Here `config2` represents the new global network configuration. The per-packet update library analyzes the configuration and topology and selects a suitable mechanism to implement the update. Note that the programmer does not write any tricky code, does not need to consider how to synchronize switch update commands, and does not need to consider the packets

in flight across the network. The `per_packet_update` library handles all of the low-level details, and even attempts to select a mechanism that minimizes the cost of implementing the update.

To implement the update, the library could use the safe, switch-update ordering described above. However, in general, it is not always possible to find such an ordering. Nevertheless, one can always achieve a per-packet consistent update using a two-phase update supported by configuration versioning. Intuitively, this universal update mechanism works by stamping every incoming packet with a version number (*e.g.*, stored in a VLAN tag) and modifying every configuration so that it only processes packets with a set version number. To change from one configuration to the next, it first populates the switches in the middle of the network with new configurations guarded by the next version number. Once that is complete, it enables the new configurations by installing rules at the perimeter of the network that stamp packets with that next version number. Though this general mechanism is somewhat heavyweight, our libraries identify and apply lightweight optimizations.

This short example illustrates some of the challenges that arise when implementing a network update with strong semantic guarantees. However, it also shows that all of these complexities can be hidden from the programmer, leaving only the simplest of interfaces for global network update. We believe this simplicity will lead to a more reliable and secure network infrastructure. The following sections describe our approach in more detail.

5.3 The Network Model

This section presents a simple mathematical model of the essential features of SDNs. This model is defined by a relation that describes the fine-grained, step-by-step execution of a

Bit	b	$::= 0 \mid 1$
Packet	pk	$::= [b_1, \ldots, b_k]$
Port	p	$::= 1 \mid \cdots \mid k \mid \mathit{Drop} \mid \mathit{World}$
Located Pkt	lp	$::= (p, pk)$
Trace	t	$::= [lp_1, \ldots, lp_n]$
Update	u	$\in \mathit{LocatedPkt} \rightharpoonup \mathit{LocatedPkt\ list}$
Switch Func.	S	$\in \mathit{LocatedPkt} \to \mathit{LocatedPkt\ list}$
Topology Func.	T	$\in \mathit{Port} \to \mathit{Port}$
Port Queue	Q	$\in \mathit{Port} \to (\mathit{Packet} \times \mathit{Trace})\ \mathit{list}$
Configuration	C	$::= (S, T)$
Network State	N	$::= (Q, C)$

(a)

T-PROCESS

if p is any port (1)
and $Q(p) = [(pk_1, t_1), (pk_2, t_2), \ldots, (pk_j, t_j)]$ (2)
and $C = (S, T)$ (3)
and $S(p, pk_1) = [(p'_1, pk'_1), \ldots, (p'_k, pk'_k)]$ (4)
and $T(p'_i) = p''_i$, for i from 1 to k (5)
and $t'_1 = t_1 +\!\!+ [(p, pk_1)]$ (6)
and $Q'_0 = \mathit{override}(Q, p \mapsto [(pk_2, t_2), \ldots, (pk_j, t_j)])$ (7)
and $Q'_1 = \mathit{override}(Q'_0, p''_1 \mapsto Q(p''_1) +\!\!+ [(pk'_1, t'_1)])$
$\vdots$
and $Q'_k = \mathit{override}(Q'_{k-1}, p''_k \mapsto Q(p''_k) +\!\!+ [(pk'_k, t'_1)])$
then $(Q, C) \longrightarrow (Q'_k, C)$ (8)

T-UPDATE

if $S' = \mathit{override}(S, u)$ (9)
then $(Q, (S, T)) \xrightarrow{u} (Q, (S', T))$ (10)

(b)

Figure 5.2: The network model: (a) syntax and (b) semantics.

network. We write the relation using the notation $N \xrightarrow{us}{}^{\star} N'$, where N is the network at the beginning of an execution, N' is the network after some number of steps of execution, and *us* is a list of "observations" that are made during the execution.[1] Intuitively, an observation should be thought of as a message between the controller and the network. In this chapter, we are interested in a single kind of message—a message u that directs a particular switch in the network to update its forwarding table with some new rules. The formal system could easily be augmented with other kinds of observations, such as topology changes or failures. For the sake of brevity, we elide these features in this chapter.

The main purpose of the model is to compute the *traces*, or paths, that a packet takes through a network that is configured in a particular way. These traces in turn define the properties, be they access control or connectivity or others, that a network configuration satisfies. Our end goal is to use this model and the traces it generates to prove that, when we update a network, the properties satisfied by the initial and final configurations are preserved. The rest of this section will make these ideas precise.

Notation We use standard notation for types. In particular, the type $T_1 \to T_2$ denotes the set of total functions that take arguments of type T_1 and produce results of type T_2, while $T_1 \rightharpoonup T_2$ denotes the set of partial functions from T_1 to T_2; the type $T_1 \times T_2$ denotes the set of pairs with components of type T_1 and T_2; and T *list* denotes the set of lists with elements of type T.

We also use standard notation to construct tuples: (x_1, x_2) is a pair of items x_1 and x_2. For lists, we use the notation $[x_1, \ldots, x_n]$ for the list of n elements x_1 through x_n, $[\,]$ for the empty list, and $xs_1 +\!\!+ xs_2$ for the concatenation of the two lists xs_1 and xs_2. Notice that

[1] When a network takes a series of steps and there are no observations (*i.e.,* no updates happen), we omit the list above the arrow, writing $N \longrightarrow^{\star} N'$ instead.

if x is some sort of object, we will typically use xs as the variable for a list of such objects. For example, we use u to represent a single update and us to represent a list of updates.

Basic Structures Figure 5.2(a) defines the syntax of the elements of the network model. A *packet pk* is a sequence of bits, where a *bit b* is either 0 or 1. A *port p* represents a location in the network where packets may be waiting to be processed. We distinguish two kinds of ports: ordinary ports numbered uniquely from 1 to k, which correspond to the physical input and output ports on switches, and two special ports, *Drop* and *World*. Intuitively, packets queued at the *Drop* port represent packets that have been dropped, while packets queued at the *World* port represent packets that have been forwarded beyond the confines of the network. Each ordinary port will be located on some switch in the network. However, we will leave the mapping from ports to switches unspecified, as it is not needed for our primary analyses.

Switch and Topology Functions A network is a packet processor that forwards packets and optionally modifies the contents of those packets on each hop. Following Kazemian *et al.* [49], we model packet processing as the composition of two simpler behaviors: (1) forwarding a packet across a switch and (2) moving packets from one end of a link to the other end. The *switch function* S takes a located packet lp (a pair of a packet and a port) as input and returns a list of located packets as a result. In many applications, a switch function only produces a single located packet, but in applications such as multicast, it may produce several. To drop a packet, a switch function maps the packet to the special *Drop* port. The *topology function* T maps one port to another if the two ports are connected by a link in the network. Given a topology function T, we define an ordinary port p to be an *ingress port* if for all other ordinary ports p' we have $T(p') \neq p$. Similarly, we define an

ordinary port p to be an *internal port* if it is not an ingress port.

To ensure that switch and topology functions are reasonable, we impose the following conditions:

(1) For all packets pk, $S(Drop, pk) = [(Drop, pk)]$ and
$S(World, pk) = [(World, pk)]$;

(2) $T(Drop) = Drop$ and $T(World) = World$; and

(3) For all ports p and packets pk
if $S(p, pk) = [(p_1, pk_1), \ldots, (p_k, pk_k)]$ we have $k \geq 1$.

Taken together, the first and second conditions state that once a packet is dropped or forwarded beyond the perimeter of the network, it must stay dropped or beyond the perimeter of the network and never return. If our network forwards a packet out to another network and that other network forwards the packet back to us, we treat the return packet as a "fresh" packet—*i.e.*, we do not explicitly model inter-domain forwarding. The third condition states that applying the forwarding function to a port and a packet must produce at least one packet. This third condition means that the network cannot drop a packet simply by not forwarding it anywhere. Dropping packets occurs by explicitly forwarding a single packet to the *Drop* port. This feature makes it possible to state network properties that require packets either be dropped or not.

Configurations and Network States A *trace* t is a list of located packets that keeps track of the hops that a packet takes as it traverses the network. A *port queue* Q is a total function from ports to lists of packet-trace pairs. These port queues record the packets waiting to be processed at each port in the network, along with the full processing history

of that packet. Several of our definitions require modifying the state of a port queue. We do this by building a new function that overrides the old queue with a new mapping for one of its ports: $override(Q, p \mapsto l)$ produces a new port queue Q' that maps p to l and like Q otherwise.

$$override(Q, p \mapsto l) = Q'$$

$$\text{where } Q'(p') = \begin{cases} l & \text{if } p = p' \\ Q(p') & \text{otherwise} \end{cases}$$

A *configuration* C comprises a switch function S and a topology function T. A *network state* N is a pair (Q, C) containing a port queue Q and configuration C.

Transitions The formal definition of the network semantics is given by the relations defined in Figure 5.2(b), which describe how the network transitions from one state to the next one. The system has two kinds of transitions: packet-processing transitions and update transitions. In a packet-processing transition, a packet is retrieved from the queue for some port, processed using the switch function S and topology function T, and the newly generated packets are enqueued onto the appropriate port queues. More formally, packet-processing transitions are defined by the T-Process case in Figure 5.2(b). Lines 1-8 may be read roughly as follows:

(1) If p is any port,

(2) a list of packets is waiting on p,

(3) the configuration C is a pair of a switch function S and topology function T,

(4) the switch function S forwards the chosen packet to a single output port, or several ports in the case of multicast, and possibly modifies the packet

(5) the topology function T connects each output port to an input port,

(6) a new trace t'_1, which extends the old trace and records the current hop, is generated,

(7) a new queue Q'_k is generated by moving packets across links as specified in steps (4), (5) and (6),

(8) then (Q, C) can step to (Q'_k, C).

In an update transition, the switch forwarding function is updated with new behavior. We represent an *update* u as a partial function from located packets to lists of located packets (*i.e.*, an update is just a "part" of a global (distributed) switch function). To apply an update to a switch function, we overwrite the function using all of the mappings contained in the update. More formally, $override(S, u)$ produces a new function S' that behaves like u on located packets in the domain[2] of u, and like S otherwise.

$$override(S, u) = S'$$

$$\text{where } S'(p, pk) = \begin{cases} u(p, pk) & \text{if } (p, pk) \in \mathsf{dom}(u) \\ S(p, pk) & \text{otherwise} \end{cases}$$

Update transitions are defined formally by the T-UPDATE case in Figure 5.2(b). Lines 9-10 may be read as follows: if S' is obtained by applying update u to a switch in the network then network state $(Q, (S, T))$ can step to new network state $(Q, (S', T))$.

Network Semantics The overall semantics of a network in our model is defined by allowing the system to take an arbitrary number of steps starting from an *initial state* in which the queues of all internal ports as well as *World* and *Drop* are empty, and the queues of external ports are filled with pairs of packets and the empty trace. The reflexive and transitive closure of the single-step transition relation $N \xrightarrow{us}{}^{\star} N'$ is defined in the usual way, where the sequence of updates recorded in the label above the arrow is obtained by concatenating

[2]Domain of an update is the set of located packets it's defined upon.

all of the updates in the underlying transitions in order.[3] A network *generates* a trace t if and only if there exists an initial state Q such that $(Q, C) \longrightarrow^\star (Q', C)$ and t appears in Q'. Note that no updates may occur when generating a trace.

Properties In general, there are myriad properties a network might satisfy—*e.g.*, access control, connectivity, in-order delivery, quality of service, fault tolerance, to name a few. In this chapter, we will primarily be interested in *trace properties*, which are prefix-closed sets of traces. Trace properties characterize the paths (and the state of the packet at each hop) that an individual packet is allowed to take through the network. Many network properties, including access control, connectivity, routing correctness, loop-freedom, correct VLAN tagging, and waypointing can be expressed using trace properties. For example, loop-freedom can be specified using a set that contain all traces except those in which some ordinary port p appears twice. In contrast, timing properties and relations between multiple packets including quality of service, congestion control, in-order delivery, or flow affinity are not trace properties.

We say that a port queue Q satisfies a trace property P if all of the traces that appear in Q also appear in the set P. Similarly, we say that a network configuration C satisfies a trace property P if for all *initial* port queues Q and all (update-free) executions $(Q, C) \longrightarrow^\star (Q', C)$, it is the case that Q' satisfies P.

[3]The semantics of the network is defined from the perspective of an omniscient observer, so there is an order in which the steps occur.

5.4 Per-Packet Abstraction

One reason that network updates are difficult to get right is that they are a form of concurrent programming. Concurrent programming is hard because programmers must consider the interleaving of every operation in every thread and this leads to a combinatorial explosion of possible outcomes—too many outcomes for most programmers to manage. Likewise, when performing a network update, a programmer must consider the interleaving of switch update operations with every kind of packet that might be traversing their network. Again, the number of possibilities explodes.

Per-packet consistent updates reduce the number of scenarios a programmer must consider to just two: for every packet, it is as if the packet flows through the network *completely before* the update occurs, or *completely after* the update occurs.

One might be tempted to think of per-packet consistent updates as "atomic updates", but they are actually better than that. An atomic update would cause packets in flight to be processed partly according to the configuration in place prior to the update, and partly according to the configuration in place after the update. To understand what happens to those packets (*e.g.*, whether they get dropped), a programmer would have to reason about every possible trace formed by concatenating a prefix generated by the original configuration with a suffix generated by the new configuration.

Intuitively, per-packet consistency states that for a given packet, the traces generated during an update come from the old configurations, or the new configuration, but not a mixture of the two. In the formal definition of per-packet consistency, we introduce an equivalence relation $\sim$ on packets. We extend this equivalence relation to traces by considering two traces to be equivalent if the packets they contain are equivalent according to

the $\sim$ relation (similarly, we extend $\sim$ to properties in the obvious way). We then require that all traces generated *during* the update be equivalent to a trace generated by either the initial or final configuration. For the rest of the chapter, when we say that $\sim$ is an equivalence relation on traces, we assume that it has been constructed like this. This specification gives implementations of updates flexibility by allowing some minor, irrelevant differences to appear in traces (where $\sim$ defines the notion of irrelevance precisely). For example, we can define a "version" equivalence relation that relates packets pk and pk' which differ only in the value of their version tags. This relation will allow us to state that changes to version tags performed by the implementation mechanism for per-packet update are irrelevant. In other words, a per-packet mechanism may perform internal bookkeeping by stamping version tags without violating our technical requirements on the correctness of the mechanism. The precise definition of per-packet consistency is as follows.

Definition 4 (Per-packet $\sim$-consistent update)**.** *Let $\sim$ be a trace-equivalence relation. An update sequence us is a per-packet $\sim$-consistent update from C_1 to C_2 if and only if, for all*

- *initial states Q,*
- *executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*
- *and traces t in Q',*

there exists

- *an initial state Q_i,*
- *and either an execution $(Q_i, C_1) \longrightarrow^{\star} (Q'', C_1)$ or an execution $(Q_i, C_2) \longrightarrow^{\star} (Q'', C_2)$,*

such that Q'' contains t', for some trace t' with $t' \sim t$.

From an implementer's perspective, the operational definition of per-packet consistency given above provides a specification that he or she must meet. However, from a programmer's perspective, there is another, more useful side to per-packet consistency: *per-packet consistent updates preserve every trace property.*

Definition 5 ($\sim$-property preservation)**.** *Let C_1 and C_2 be configurations and $\sim$ be a trace-equivalence relation. A sequence us is a $\sim$-property-preserving update from C_1 and C_2 if and only if, for all*

- *initial states Q,*
- *executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*
- *and properties P that are satisfied by C_1 and C_2 and do not distinguish traces related by $\sim$,*

we have that Q' satisfies P.

Universal $\sim$-property preservation gives programmers a strong principle they can use to reason about their programs. If programmers check that a trace property such as loop-freedom or access control holds of the network configurations before and after an update, they are guaranteed it holds of every trace generated throughout the update process, even though the series of observations us may contain many discrete update steps. Our main theorem states that per-packet consistent updates preserve all properties:

Theorem 1. *For all trace-equivalence relations $\sim$, if us is a per-packet $\sim$-consistent update of C_1 to C_2 then us is a $\sim$-property-preserving update of C_1 to C_2.*

The proof of the theorem is a relatively straightforward application of our definitions. From a practical perspective, this theorem allows a programmer to get great mileage out of

per-packet consistent updates. In particular, since per-packet consistent updates preserve *all* trace properties, the programmers do not have to tell the system *which* specific properties must be invariant in their applications.

From a theoretical perspective, it is also interesting that the converse of the above theorem holds. This gives us a sort of completeness result: if programmers want an update that preserves all properties, they need not search for it outside of the space of per-packet consistent updates—any universal trace-property preserving update is a per-packet consistent update.

Theorem 2. *For all trace-equivalence relations $\sim$, if us is a $\sim$-property-preserving update of C_1 to C_2 then us is a per-packet $\sim$-consistent update of C_1 to C_2.*

The proof of this theorem proceeds by observing that since *us* preserves all $\sim$-properties, it certainly preserves the following $\sim$-property P_{or}:

$$\begin{aligned}\{t \mid &\text{there exists an initial } Q \text{ and a trace } t' \\ &\text{and } ((Q, C_1) \longrightarrow^\star (Q', C_1) \text{ or } (Q, C_2) \longrightarrow^\star (Q', C_2)), \\ &\text{and } t \sim t', \\ &\text{and } t' \in Q'\}\end{aligned}$$

By the definition of P_{or}, the update *us* generates no traces that cannot be generated either by the initial configuration C_1 or by the final configuration C_2. Hence, by definition, *us* is per-packet consistent.

Formal proof The network model, and all of the above theorems have been formally specified and proven in the Coq theorem prover.

5.5 Per-packet Mechanisms

Depending on the network topology and the specifics of the configurations involved, there may be several ways to implement a per-packet consistent update. However, all of the techniques we have discovered so far, no matter how complicated, can be reduced to two fundamental building blocks: the one-touch update and the unobservable update. For example, our two-phase update mechanism uses unobservable updates to install the new configuration before it is used, and then "unlocks" the new policy by performing a one-touch update on the ingress ports.

One-touch updates A one-touch update is an update with the property that no packet can follow a path through the network that reaches an updated (or to-be-updated) part of the switch rule space more than once.

Definition 6 (One-touch Update)**.** *Let $C_1 = (FT, T)$ be the original network configuration, $us = [u_1, \ldots, u_k]$ an update sequence, and $C_2 = (FT[u_1, \ldots, u_k], T)$ the new configuration, such that the domains of each update u_1 to u_k are mutually disjoint. If, for all*

- *initial states Q,*
- *and executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*

there does not exist a trace t in Q' such that

- *t contains distinct trace elements (p_1, pk_1) and (p_2, pk_2),*
- *and (p_1, pk_1) and (p_2, pk_2) both appear in the domain of update functions $[u_1, \ldots, u_k]$,*

then us is a one-touch update from C_1 to C_2.

Theorem 9. *If us is a one-touch update then us is a $\sim$-per-packet consistent update for any $\sim$.*

The proof proceeds by considering the possible traces t generated by an execution $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$. There are two cases: (1) There is no element of t that appears in the domain of an update function in us, or (2) some element lp of t appears in the domain of an update function in us. In case (1), t can also be generated by an execution with no update observations: $(Q, C_1) \longrightarrow^{\star} (Q'', C_1)$, and the definition of per-packet consistency vacuously holds. In case (2), there are two subcases:

(i) lp appears in the trace prior to the update taking place and so t is also generated by $(Q, C_1) \longrightarrow^{\star} (Q'', C_1)$.

(ii) lp appears in the trace after the update has taken place and so t is also generated by $(Q, C_2) \longrightarrow^{\star} (Q'', C_2)$.

The one-touch update mechanism has a few immediate, more specific applications:

- *Loop-free switch updates:* If a switch is not part of a topological loop (either before or after the update), then updating all the ports on that switch is an instance of a one-touch update and is per-packet consistent.
- *Ingress port updates:* An ingress port interfaces exclusively with the external world, so it can not be a part of an internal topological loop and is never on the same trace as any other ingress port. Consequently, any update to ingress ports is a one-touch update and is per-packet consistent. Such updates can be used to change the admission control policy for the network, either by adding or excluding flows.

When one-touch updates are combined with unobservable updates, there are many more possibilities.

Unobservable updates An unobservable update is an update that does not change the set of traces generated by a network.

Definition 7 (Unobservable Update)**.** *Let $C_1 = (FT, T)$ be the original network configuration, $us = [u_1, \ldots, u_k]$ an update sequence, and $C_2 = (FT[u_1, \ldots, u_k], T)$ the new configuration. If, for all*

- *initial states Q,*
- *executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*
- *and traces t in Q',*

there exists

- *an initial state Q_i,*
- *and an execution $(Q_i, C_1) \longrightarrow^{\star} (Q'', C_1)$,*

such that the trace t is in Q'', then us is an unobservable update from C_1 to C_2.

Theorem 3. *If us is an unobservable update then us is a per-packet consistent update.*

The proof proceeds by observing that every trace generated during the unobservable update $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$ also appears in the traces generated by C_1.

On their own, unobservable updates are useless as they do not change the semantics of packet forwarding. However, they may be combined with other per-packet consistent updates to great effect using the following theorem.

Theorem 10 (Composition)**.** *If us_1 is an unobservable update from C_1 to C_2 and us_2 is a per-packet consistent update from C_2 to C_3 then $us_1 +\!\!+ us_2$ is a per-packet consistent update from C_1 to C_3.*

A simple use of composition arises when one wants to achieve a per-packet consistent update that extends a policy with a completely new path.

- *Path extension:* Consider an initial configuration C_1. Suppose $[u_1, u_2, \ldots, u_k]$ updates ports $p_1, p_2, \ldots, p_k$ respectively to lay down a new path through the network with u_1 updating the ingress port. Suppose also that the ports updated by $us = [u_2, \ldots, u_k]$ are unreachable in network configuration C_1. Hence, us is an unobservable update. Since $[u_1]$ updates an ingress port, it is a one-touch update and also per-packet consistent. By the composition principle, $us +\!\!+ [u_1]$ is a per-packet consistent update.

Notice that the path update is achieved by first laying down rules on switches 2 to k and then, when that is complete, laying down the rules on switch 1. A well-known (but still common!) bug occurs when programmers attempt to install new forwarding paths but lay down the elements of the path in wrong order [14]. Typically, there is a race condition in which packets traverse the first link and reach the switch 2 before the program has had time to lay down the rules on links 2 to k. Then when packets reach switch 2, it does not yet know how to handle them, and a default rule sends the packets to the controller. The controller often becomes confused as it begins to see additional packets that should have already been dealt with by laying down the new rules. The underlying cause of this bug is explained with our model—the programmer intended a per-packet consistent update of the policy with a new path, but failed to implement per-packet consistency correctly. All such bugs are eradicated from network programs if programmers use per-packet consistent

updates and never use their own ad hoc update mechanisms.

Two-phase update So far, all of our update mechanisms have applied to special cases in which the topology, existing configuration, and/or updates have specific properties. Fortunately, provided there are a few bits in packets that are irrelevant to the network properties a programmer wishes to enforce, and can be used for bookkeeping purposes, we can define a mechanism that handles arbitrary updates using a two-phase update protocol.

Intuitively, the two-phase update works by first installing the new configuration on internal ports, but only enabling the new configuration for packets containing the correct version number. It then updates the ingress ports one-by-one to stamp packets with the new version number. Notice that the updates in the first phase are all unobservable, since before the update, the ingress ports do not stamp packets with the new version number. Hence, since updating ingress ports is per-packet consistent, by the composition principle, the two-phase update is also per-packet consistent.

To define the two-phase update formally, we need a few additional definitions. Let a version-property be a trace property that does not distinguish traces based on the value of version tags. A configuration C is a version-n configuration if $C = (S, T)$ and S modifies packets processed by any ingress port p_{in} so that after passing through p_{in}, the packet's version bit is n. We assume that the S function does not otherwise modify the version bit of the packet. Two configurations C and C' coincide internally on version-n packets whenever $C = (S, T)$ and $C' = (S', T')$ and for all internal ports p, and for all packets pk with version bit set to n, we have that $FT(p, pk) = FT'(p, pk)$. Finally, an update u is a refinement of S, if for all located packets lp in the domain of u, we have that $u(lp) = S(lp)$.

Definition 8 (Two-phase Update)**.** *Let* $C_1 = (S, T)$ *be a version-*1 *configuration and* $C_2 =$

(S', T) be a version-2 configuration. Assume that C_1 and C_2 coincide internally on version-1 packets. Let $us = [u_1^i, \ldots, u_m^i, u_1^e, \ldots, u_n^e]$ be an update sequence such that

- *$S' = override(S, us)$,*
- *each u_j^i and u_k^e is a refinement of S',*
- *p is internal, for each (p, pk) in the domain of u_j^i,*
- *and p is an ingress, for each (p, pk) in the domain of u_k^e.*

Then us is a two-phase update from C_1 to C_2.

Theorem 11. *If us is a two-phase update then us is per-packet consistent.*

The proof simply observes that $us_1 = [u_1^i, \ldots, u_m^i]$ is an unobservable update, and $us_2 = [u_1^e, \ldots, u_n^e]$ is a one-touch update (and therefore per-packet consistent). Hence, by composition, the two-phase update $us_1 +\!\!+ us_2$ is per-packet consistent.

Optimized mechanisms Ideally, update mechanisms should satisfy *update proportionality*, where the cost of installing a new configuration should be proportional to the size of the configuration change. A perfectly proportional update would (un)install just the "delta" between the two configurations. The full two-phase update mechanism that installs the full new policy and then uninstalls the old policy lacks update proportionality. In this section, we describe optimizations that substantially reduce overhead.

Pure extensions and retractions are one important case of updates where a per-packet mechanism achieves perfect proportionality. A pure extension is an update that adds new paths to the current configuration that cannot be reached in the old configuration—*e.g.*, adding a forwarding path for a new host that comes online. Such updates do not require a

complete two-phase update, as only the new forwarding rules need to be installed—first at the internal ports and then at the ingresses. The rules are installed using the current version number. A *pure retraction* is the dual of a pure extension in which some paths are removed from the configuration. Again, the paths being removed must be unreachable in the new configuration. Pure retractions can be implemented by updating the ingresses, pausing to wait until packets in flight drain out of the network, and then updating the internal ports.

If paths are not only added or removed but are modified then more powerful optimizations are available. Per-packet consistency requires that the active paths in the network come from either of the configurations. The subset mechanism works by identifying the paths that have been added, removed or changed and then updating the rules along the entire path to use a new version. This optimization is always applicable, but in the degenerate case it devolves into a network-wide two-phase update.

5.6 Per-flow Consistency

Per-packet consistency, while simple and powerful, is not always enough. Some applications require a stream of related packets to be handled consistently. For example, a server load-balancer needs all packets from the same TCP connection to reach the same server replica. In this section, we introduce the per-flow consistency abstraction, and discuss mechanisms for per-flow consistent updates.

Per-flow abstraction To see the need for per-flow consistency, consider a network where a single switch S load-balances between two back-end servers A and B. Initially, S directs traffic from IP addresses starting with 0 (*i.e.*, source addresses in 0.0.0.0/1) to A and 1 (*i.e.*, source addresses in 128.0.0.0/1) to B. At some time later, we bring two additional servers C

and D online, and re-balance the load using a two-bit prefix, directing traffic from addresses starting with 00 to A, 01 to B, 10 to C, and 11 to D.

Intuitively, we want to process packets from new TCP connections according to the new configuration. However, all packets in existing flows must go to the same server, where a *flow* is a sequence of packets with related header fields, entering the network at the same port, and not separated by more than n seconds. The particular value of n depends upon the protocol and application. For example, the switch should send packets from a host whose address starts with "11" to B, and not to D as the new configuration would dictate, if the packets belong to an ongoing TCP connection. Simply processing individual packets with a single configuration does not guarantee the desired behavior.

Per-flow consistency guarantees that all packets in the same flow are handled by the same version of the configuration. Formally, the per-flow abstraction preserves all path properties, as well as all properties that can be expressed in terms of the paths traversed by *sets* of packets belonging to the same flow.

Per-flow mechanisms Implementing per-flow consistent updates is much more complicated than per-packet consistency because the system must identify packets that belong to active flows. Below, we discuss three different mechanisms. Our system implements the first of the three; the latter two, while promising, depend upon technology that is not yet available in OpenFlow.

Switch rules with timeouts: A simple mechanism can be obtained by combining versioning with rule timeouts, similar to the approach in [105]. The idea is to pre-install the new configuration on the internal switches, leaving the old version in place, as in per-packet consistency. Then, on ingress switches, the controller sets soft timeouts on the rules for the

old configuration and installs the new configuration at lower priority. When all flows matching a given rule finish, the rule automatically expires and the rules for the new configuration take effect. When multiple flows match the same rule, the rule may be artificially kept alive even though the "old" flows have all completed. If the rules are too coarse, then they may never die! To ensure rules expire in a timely fashion, the controller can refine the old rules to cover a progressively smaller portion of the flow space. However, "finer" rules require more rules, a potentially scarce commodity. Managing the rules and dynamically refining them over time can be a complex bookkeeping task, especially if the network undergoes a *subsequent* configuration change before the previous one completes. However, this task can be implemented and optimized once in a run-time system, and leveraged over and over again in different applications.

Wildcard cloning: An alternative mechanism exploits the wildcard *clone* feature of the DevoFlow extension of OpenFlow [73]. When processing a packet with a *clone* rule, a DevoFlow switch creates a new "microflow" rule that matches the packet header fields exactly. In effect, clone rules cause the switch to maintain a concrete representation of each active flow. This enables a simple update mechanism: first, use clone rules whenever installing configurations; second, to update from old to new, simply replace all old clone rules with the new configuration. Existing flows will continue to be handled by the exact-match rules previously generated by the old clone rules, and new flows will be handled by the new clone rules, which themselves immediately spawn new microflow rules. While this mechanism does not require complicated bookkeeping on the controller, it does require a more complex switch.

End-host feedback: The third alternative exploits information readily available on the end hosts, such as servers in a data center. With a small extension, these servers could provide a list of active sockets (identified by the "five tuple" of IP addresses, TCP/UDP

ports, and protocol) to the controller. As part of performing an update, the controller would query the local hosts and install high-priority microflow rules that direct each active flow to the assigned server replica. These rules could "timeout" after a period of inactivity, allowing future traffic to "fall through" to the new configuration. Alternatively, the controller could install "permanent" microflow rules, and explicitly remove them when the socket no long exists on the host, obviating the need for any assumptions about the minimum interval time between packets of the same connection.

5.7 Update Mechanisms

Ideally, update mechanisms should satisfy *update proportionality*, the where the cost of installing a new configuration should be proportional to the size of the configuration change. A perfectly proportional update would (un)install just the "delta" between the two configurations. The full two-phase update mechanism that installs the full new policy and then uninstalls the old policy lacks update proportionality. In this section, we describe optimizations that substantially reduce overhead.

Pure extensions and retractions are one important case of updates where a per-packet mechanism achieves perfect proportionality. A pure extension is an update that adds new paths to the current configuration that cannot be reached in the old configuration—*e.g.*, adding a forwarding path for a new host that comes online. Such updates do not require a complete two-phase update, as only the new forwarding rules need to be installed—first at the internal ports and then at the ingresses. The rules are installed using the current version number. A *pure retraction* is the dual of a pure extension in which some paths are removed from the configuration. Again, the paths being removed must be unreachable in the new configuration. Pure retractions can be implemented by updating the ingresses, pausing to

wait until packets in flight drain out of the network, and then updating the internal ports.

If paths are not only added or removed but are modified then more powerful optimizations than pure extension/retraction are available. Per-packet consistency requires that the active paths in the network come from either of the configurations. There are two different ways to perform an update that maintains this promise. Either you identify the paths that have been added, removed, or changed and you update the entire path to use a new version, or you identify the switches that have changed and you update the entire switch to use a new version. We call the former mechanism the "subset" mechanism and the latter the "island" mechanism. Both of these mechanisms are always safe to apply but require analysis on the configurations. In the degenerate case, these mechanisms devolve to a network-wide two-phase update for all traffic.

Subset Mechanism The *subset* mechanism calculates the precise set of forwarding paths affected by an update and only updates the portion of the configuration that implements those paths (using a standard two-phase update). In situations where the set of paths is small compared to the size of the overall configuration, the cost of a subset update is less than a full two-phase update. Unlike pure extensions and retractions, which do not handle cases where existing forwarding paths are modified or where the affected rules are reachable in the new configuration, this mechanism can always be safely applied; in the case where every rule is affected by the update, it simply degenerates to a two-phase update.

The subset mechanism is implemented by computing the set of rules that changed in the new configuration and computing the closure of that set under a certain connectivity relation. We say that rules r_1, r_2 are connected under $C = (S, T)$ if there are packets pk, pk' and ports p, p' such that $r_1((p, pk)) = [..., (p', pk'), ...]$ and $(T(p'), pk') \in \mathrm{dom}(r_2)$. Write

$r_1 \leftarrow C \rightarrow r_2$ if r_1 is connected to r_2 under C or vice versa.

Definition 9 (Subset Update)**.** *Let $C = (S, T)$ be a version-*1 *configuration and $C' = (S', T)$ be a version-*2 *configuration. Let $mods_0 = S' - S$, and let mods be the closure of $mods_0$ under the relation $\bullet \leftarrow C' \rightarrow \bullet$. Then mods is a* subset *update from C to C'.*

Theorem 12. *A subset update from $C = (S, T)$ to $C' = (S', T)$ is a per-packet consistent update from C to C'.*

Proof Sketch: First show that if a set of rules is closed under the connectivity relation, then there is a well-defined set of traces generated by that set of rules, and that set of traces is equivalent to running the whole configuration over packets in the domain of the set. Then show that $S[mods] = S'$.

Island Mechanism The key idea behind the *island* mechanism is to identify a small connected component of the network containing the switches whose configurations changed. We update this "island" of switches to use the new configuration with a new version number. The configurations use a new version when packets are sent between switches in the island and restores the old version when packets leave the island. If more than two versions are active in the network at a time then versions should be implemented with a mechanism that supports a stack of versions. MPLS labels, available in OpenFlow 1.1 are a candidate.

The computation of an *island* update uses a closure computation similar to the *subset* mechanism, except that the relation is over ports instead of rules. Say that two ports p, p'' are connected under $C = (S, T)$ via a third port p' if there exists a sequence of rules $r_1, \ldots, r_k, \ldots r_n$ such that $r_1 \in S[p]$, $r_k \in S[p']$, $r_n \in S[p'']$ and for $0 < i < n$, $r_i \leftarrow C \rightarrow r_{i+1}$. Write $p \leftarrow C, p' \rightarrow p''$ if p, p'' are connected under C via p'.

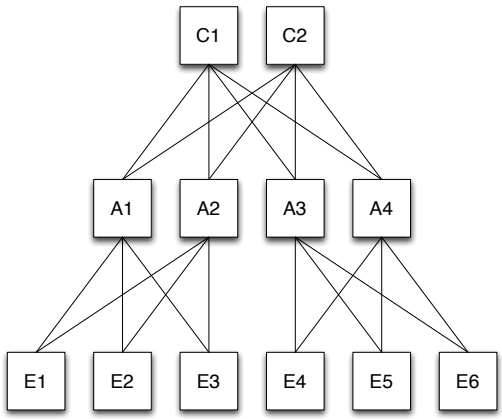


Figure 5.3: Fat tree topology

Definition 10 (Island Update)**.** *Let* $C = (S, T)$ *be a version-*1 *configuration and* $C' = (S', T)$ *be a version-*2 *configuration. Let* $mods_0 = \{p \mid p \in S' - S\}$*. Let mods be the least fixpoint of:*

$$mods = \{p \mid p \in mods_0\} \cup \{p' \mid \exists p', p'' \in mods.\ p' \leftarrow C', p \rightarrow p''\}$$

Then mods is an island *update from* C *to* C'*.*

5.7.1 Case Study

To highlight the uses and distinctions between the mechanisms, we work through case studies of networks in a fat tree and a small-world topology.

We show a simple fat tree topology and a snippet of the routing configuration in Figure 5.3. The topology consists of edge switches E1-E6 directly connected to hosts on per-switch subnets, aggregation switches A1-A4, and core switches C1-C2 providing connectivity be-

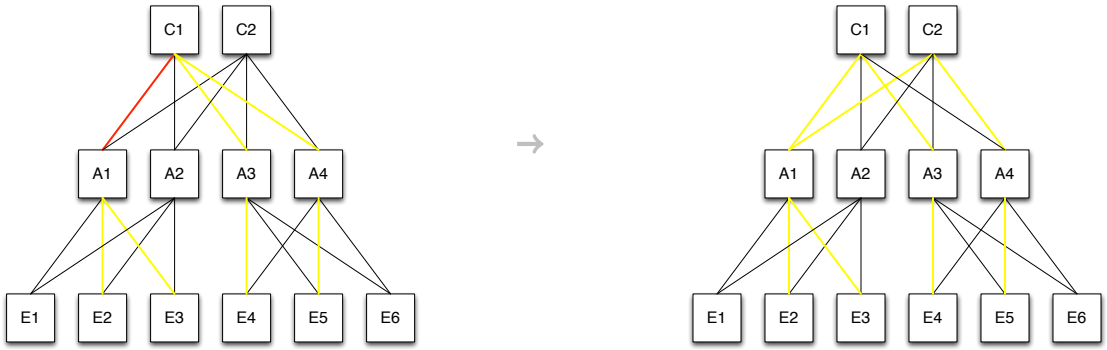


Figure 5.4: Network before and after load balancing

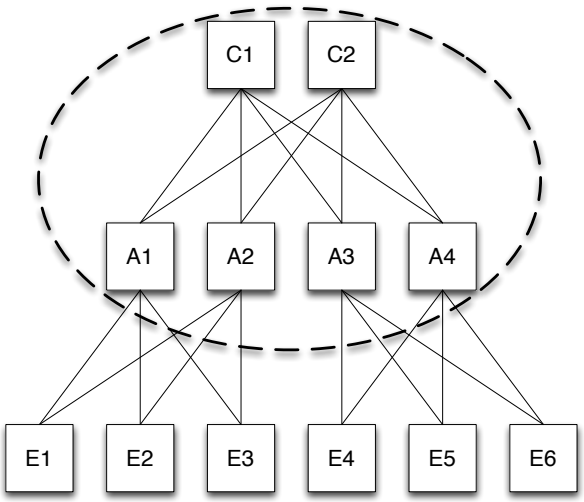


Figure 5.5: Island calculated for maintenance update

tween the two sides of the tree. The controller program runs a simple shortest path routing algorithm and monitors the network to perform load balancing. In addition, the controller takes certain switches down at predetermined times for scheduled maintenance.

Dynamic Host When a host comes on or offline at an edge switch, the routes for all the other source-destination pairs remains unchanged. Consider a scenario in which a new host H0 comes online at E1. Routes to and from H0 are installed at each switch, but existing rules and traffic are unaffected. Using a full two-phase update requires updating the configuration of every other switch, a gross violation of proportionality. Instead, the *extension* mechanism installs just the new forwarding rules at the current version number and leaves the existing rules untouched.

Network Load Balancing Initially, traffic between the two halves of the network is statically split evenly between C1 and C2, but the dynamic traffic patterns may make this static split unbalanced. Consider what happens if two pairs of hosts start sending heavy traffic flows over C1's link to A1, overloading it as shown in Figure 5.4. The controller program moves to a new configuration that puts the traffic from one of these pairs onto C2 to balance the load. Using a full update would require reinstalling all the rules in the network at the new version, even rules independent of the change. The *subset* mechanism instead updates just the rules involving the affected pair of hosts.

Switch Maintenance For scheduled maintenance, the controller installs a configuration that removes traffic from C1 and puts it all onto C2. The *island mechanism* recognizes that the aggregation and core switches form a connected component that contains all switches affected by the update and restricts the update to just that subgraph of the network, shown in Figure 5.5.

Application	Toplogy	Update	2PC		Subset		
			Ops	*Max Overhead*	*Ops*	*Ops %*	*Max Overhead*
Routing	Fat Tree	Hosts	239830	92%	119003	50%	20%
		Routes	266234	100%	123929	47%	10%
		Both	239830	92%	142379	59%	20%
	Waxman	Hosts	273514	88%	136230	49%	66%
		Routes	299300	90%	116038	39%	9%
		Both	267434	91%	143503	54%	66%
	Small World	Hosts	320758	80%	158792	50%	30%
		Routes	326884	85%	134734	41%	23%
		Both	314670	90%	180121	57%	41%
Multicast	Fat Tree	Hosts	1043	100%	885	85%	100%
		Routes	1170	100%	634	54%	57%
		Both	1043	100%	949	91%	100%
	Waxman	Hosts	1037	100%	813	78%	100%
		Routes	1132	85%	421	37%	50%
		Both	1005	100%	821	82%	100%
	Small World	Hosts	1133	100%	1133	100%	100%
		Routes	1114	90%	537	48%	66%
		Both	1008	100%	1008	100%	100%

Experimental results comparing two-phase update (2PC) with our subset optimization (Subset). We add or remove hosts and change routes to trigger configuration updates. The *Ops* column measures the number of OpenFlow install operations used in each situation. The Subset portion of the table also has an additional column (Ops %) that tabulates (Subset Ops / 2PC Ops). *Overhead* measures the extra rules concurrently installed on a switch by our update mechanisms. We pessimistically present the maximum of the overheads for all switches in the network – there may be many switches in the network that never suffer that maximum overhead.

Table 5.2: Experimental results.

5.8 Implementation and Evaluation

We have built a system called Kinetic that implements the update abstractions introduced in this chapter, and evaluated its performance on small but canonical example applications. This section summarizes the key features of Kinetic and presents experimental results that quantify the cost of implementing network updates in terms of the number of rules added and deleted on each switch.

Implementation overview Kinetic is a run-time system that sits on top of the NOX OpenFlow controller [34]. The system comprises several Python classes for representing network configurations and topologies, and a library of update mechanisms. The interface to these mechanisms is through the `per_packet_update` and `per_flow_update` functions. These functions take a new configuration and a network topology, and implement a transition to the new configuration while providing the desired consistency level. Both functions are currently based on the two-phase update mechanism, with the `per_flow_update` function using timeouts to track active flows. In addition to this basic mechanism, we have implemented a number of optimized mechanisms that can be applied under certain conditions—*e.g.*, when the update only affects a fraction of the network or network traffic. The runtime automatically analyzes the new configuration and topology and applies these optimizations when possible to reduce the cost of the update.

As described in Section 5.5, the two-phase update mechanism uses versioning to isolate the old configuration and traffic from the updated configuration. Because Kinetic runs on top of OpenFlow 1.0, we currently use the VLAN field to carry version tags (other options, like MPLS labels, are available in newer versions of OpenFlow). Our algorithms analyze the network topology to determine the ingress and internal ports and perform a two-phase update.

Experiments To evaluate the performance of Kinetic, we developed a suite of experiments using the Mininet [37] environment. Because Mininet does not offer performance fidelity or resource isolation between the simulated switches and the controller, we did not measure the time needed to implement an update. However, as a proxy for elapsed time, we counted the total number of install OpenFlow messages needed to implement each update, as well as the number of extra rules (beyond the size of either the old or new configurations) installed on

a switch.

To evaluate per-packet consistency, we have implemented two canonical network applications: routing and multicast. The routing application computes the shortest paths between each host in the topology and updates routes as hosts come online or go offline and switches are brought up and taken down for maintenance. The multicast application divides the hosts evenly into two multicast groups and implements IP multicast along a spanning tree that connects all of the hosts in a group. To evaluate the effects of our optimizations, we ran both applications on three different topologies each containing 192 hosts and 48 switches in each of three different scenarios. The topologies were chosen to represent realistic and proposed network topologies found in datacenters (fattree, small-world), enterprises (fattree) and a random topology (waxman). The three scenarios can be divided up into:

1. Dynamic hosts and static routes
2. Static hosts and dynamic routes
3. Dynamic hosts and dynamic routes

In each scenario, we moved between 3 different configurations, changing the network in a well-prescribed manner. In the dynamic host scenario, we randomly selected between $10\% - 20\%$ of the hosts and added or removed them from the network. In the dynamic routes scenario, we randomly selected 20% of the routes in the network, and forced them to re-route as if one of the switches in the route had been removed. For the multicast example, we changed one of the multicast groups each time. Static means that we did not change the host or routes.

To evaluate per-flow updates, we developed a load-balancing application that divides traffic between two server replicas, using a hash computed from the client's IP address.

The update for this experiment involved bringing several new server replicas online and re-balancing the load among all of the servers.

Results and analysis Table 5.2 compares the performance of the subset optimization to a full two-phase update. Extension updates are not shown: whenever an extension update is applicable, our subset mechanism performs the same update with the same overhead. The two-phase update has high overhead in all scenarios.

We subject each application to a series of topology changes—adding and dropping hosts and links—reflecting common network events that force the deployment of new network configurations. We measure the number of OpenFlow operations required for the deployment of the new configuration, as well as the overhead of installing extra rules to ensure per-packet consistency. The overhead is the ratio of the number of extra rules installed during the per-packet update of a switch divided by the (maximum) number of rules in the old or new configuration. For example, if the old and new configurations both had 100 rules and during the update the switch had 120 rules installed, that would be a 20% overhead. The *Overhead* column in Table 5.2 presents the maximum overhead of all switches in the network. Two-phase update requires approximately 100% overhead, because it leaves the old configuration on the switch as it installs the new one. Because both configurations may not be precisely the same size, it is not always exactly 100%. In some cases, the new configuration may be much smaller than the old (for example, when routes are diverted away from a switch) and the overhead is much lower than 100%.

The first routing scenario, where hosts are added or removed, demonstrates the potential of our optimizations. When a new host comes online, the application computes routes between it and every other online host. Because the rules for the new routes do not affect

traffic between existing hosts, they can be installed without modifying or reinstalling the existing rules. Similarly, when a host goes offline, only the installed rules routing traffic to or from that host need to be uninstalled. This leads to update costs proportional to the number of rules that changed between configurations, as opposed to a full two-phase update, where the cost is proportional to the size of the entire new configuration.

Our optimizations yield fewer improvements for the multicast example, due to the nature of the example: when the spanning tree changes, almost all paths change, triggering an expensive update.

We have not applied our optimizations to the per-flow mechanism, therefore we do not include an optimization evaluation of the load balancing application.

5.9 Conclusions and Future Work

Reasoning about concurrency is notoriously difficult, and network software is no exception. To make fundamental progress, the networking field needs simple, general, and reusable abstractions for changing the configuration of the network. Our per-packet and per-flow consistency abstractions allow programmers to focus their attention on the state of the network before and after a configuration change, without worrying about the transition in between. The update abstractions are powerful, in that the programmer does not need to identify the properties that should hold during the transition, since *any* property common to both configurations holds for *any* packet traversing the network during the update. This enables lightweight verification techniques that simply verify the properties of the old and new configurations. In addition, our abstractions are practical, in that efficient and correct update mechanisms exist and are implementable using today's OpenFlow switches. Our

implementation and Coq proofs are available at our website www.frenetic-lang.org.

In our ongoing work, we are exploring new mechanisms that make network updates faster and cheaper, by limiting the number of rules or the number of switches affected. In this investigation, our theoretical model is a great asset, enabling us to prove that our proposed optimizations are correct. We also plan to extend our formal model to capture the per-flow consistent update abstraction, and prove the correctness of the per-flow update mechanisms. In addition, we will make our update library available to the community, to enable future OpenFlow applications to leverage these update abstractions. Finally, while per-packet consistency and per-flow consistency are core abstractions with excellent semantic properties, we want to explore other notions of consistency that either perform better (but remain sufficiently strong to provide benefits beyond eventual consistency) or provide even richer guarantees.

CHAPTER 6

RELATED WORK

6.1 General approaches

This thesis proposes a methodology for building reliable networking systems through verification. This is not the first such proposal; see *e.g.*the survey paper [83]. Indeed, a system called *Formally Verifiable Networking* by Wang *et al.* [104] proposed that network protocols be designed and written in a specialized logic programming language called Network Datalog (NDlog) which could then be formally analysed against a formal specification. This is similar in spirit to the verification performed in Chapter 3, though the focus and design is different. NDlog was focused upon building distributed protocols on top of a clean-slate architecture built upon Datalog. It also used the PVS theorem prover [81], which requires users to manually construct proofs of correctness, unlike the fully automated decision procedure in this thesis.

The approach taken in this thesis is to start with network programs written in domain-specific, which are usually generated by a higher-level application written in a general purpose programming language. By performing verification on the output, we essentially "cut off" the need to reason about this higher-level application to assure correctness. An alternative approach is to instead push the reasoning up the stack, and develop higher-level and more expressive primitives for network programming to enable the programmer to reason directly about the top-level program. NDlog is one example of this; another Datalog based language is FlowLog [79],[78]. FlowLog provides a *tierless* programming model in which there is a single unified abstraction shared between the control-plane, data-plane, and controller state. In addition, FlowLog policies can be verified with the Alloy analyzer [43]. Alloy is not a

complete procedure; it only performs bounded verification, but the authors of FlowLog found that bounded verification sufficed for almost all of their examples.

The Kinetic system [53][1] approaches the problem of building dynamic network systems with a domain specific language based finite-state machines that react to network events, and a temporal-logic based specification language that is used to analyze the correctness of the finite-state machines.

Verdi, [106] is a system in a similar spirit to this thesis, but focused on the distributed systems domain. Verdi is a formal framework for designing and implementing distributed systems in the Coq theorem prover. It focuses reasoning about the behavior of systems under different failure models, and allows programmers to pick and choose which failure models to adopt.

6.2 Formally verified systems

Verification technology has progressed dramatically in the past decades, making it feasible to prove useful theorems about real systems including databases [66], compilers [58], and even whole operating systems [54]. Compilers have been particularly fruitful targets for verification efforts [40]. Most prominently, the CompCert compiler translates programs in a large subset of C to PowerPC, ARM, and x86 executables [58]. The Verified Software Toolchain project provides machine-checked infrastructure for connecting properties obtained by program analysis to guarantees at the machine level [5]. Rocksalt verifies a tool for analyzing machine code against a detailed model of x86 [76]. Another system, Bedrock provides rich Coq libraries for verifying low-level programs [17]. Significant portions of

[1]Note: the system in the paper Chapter 5 is based upon was also named Kinetic. There is no relation between the two.

many other compilers have been formalized and verified, including the LLVM intermediate representation [113], the F* typechecker [96], and an extension of CompCert with garbage collection [68].

The seL4 microkernel [54] project built the first fully verified general purpose OS kernel. They started with a formal specification of full functional correctness, written in Isabelle/HOL [80], built an executable specification (written in Haskell) that provably refined the original specification, and finally built a high-performance implementation in C, which in turn refined the Haskell specification. The final seL4 system achieved similar performance to other L4 micro-kernels, but with a formal guarantee of correctness and reliability.

The CompCert project [58] is another celebrated fully formally verified system. CompCert is a C compiler built in the Coq theorem prover and fully verified for correctness against a formal model of C semantics and the semantics of the x86, ARM, and PowerPC architectures. CompCert includes high-performance, fully verified compiler optimizations, and achieves respectable performance against other, unverified compilers. In one study of compiler bugs, every compiler except the verified CompCert compiler was found to have contain bugs that caused wrong code generation [108][2]. Before CompCert, the so-called "CLInc stack" [11] was a formally verified system consisting of a high-level programming language with a verification engine (Gypsy [32]); a verified compiler for a restricted language [109]; a verified assembler (Piton [75]); a verified multitasking operating system (Kit[10]); and a fully verified microprocessor (the FM8502 [42]).

In a different vein, there have been several projects that built formal models of networks or network protocols (see *e.g.*[71] [107]). One of the most detailed formal networking models

[2]They initially found a bug in an unverified component of CompCert. In response to the bug, the project extended the verification to include that component, and the authors were no longer able to find any bugs in the compiler.

ever built is Bishop *et al.*'s model of the TCP protocol and the Sockets API[13]. They built a fully formal, highly precise model of the TCP protocol and its associated Sockets API in HOL4 [33], and exhaustively validated it against de facto reference implementations.

A portion of the PANE [21] compiler was formalized in Coq, but since the proof did not model several subtleties of flow tables, the compiler still had bugs. Unlike our system, PANE does not model or verify any portion of its run-time system. We used some of the PANE proofs during early development of our system.

6.3 Network verification tools

Specification languages Before SDN, abstract network specifications independent from implementation were largely limited to firewall policies. See, *e.g.* [7] for an entity-relationship modeling framework for specifying security policies. One particularly notable early work in this area was Guttman's filtering postures [36], which took a global network access control policy and automatically specialized it into local filters whose combination was guaranteed to enforce the global policy.

More recently, the VeriFlow [52] network verification system includes a general API for checking application-specific network invariants. This API is not exactly a specification language (it lacks semantics), and it's not exactly clear what properties it can and can not express. Invariants must be checkable in a sort of incremental manner, where only modified equivalence classes of network rules are given at each step.

The NetPlumber [48] system includes a specification language for relating network flows to allowable paths. This language is based upon regular expressions with wildcards, and is quite similar in spirit to the Pathetic language in this chapter. However, the languages

are subtly different: whereas Pathetic is based on regular expressions in the sense of formal language theory, and comes with a formal semantics, NetPlumber's language is based on regular expressions in the sense of the string matching functions found in many programming languages. Arguably, the latter design decision makes them easier to understand and more familiar to the average programmer. They also lack a semantics, making it difficult or impossible for a programmer to reason about the exact meaning of their specification.

ConfigChecker [3] is a system for analyzing firewall and router configurations. It uses specifications of network behavior specified in Computational Tree Logic (CTL), a branching-time temporal logic.

Network verifiers/static analyzers One of the first static analyzers to achieve widespread adoption was Feamster and Balakrishnan's routing configuration checker *rcc*[20]. *rcc* was a tool that statically analyzed Border Gateway Protocol (BGP) router configurations for common configuration mistakes such as typos, learning unusable paths, sharing unusable paths, or failing to learn all usable paths. *rcc* was reportedly widely welcomed in industry as an improvement over the status quo of running configurations in small test-beds and then pushing them out to production to detect bugs. *rcc* was limited to detecting generic configuration faults, and was not able to verify full application specific functional correctness.

Recent years have seen an incredible interest in the development of verification tools for the networking domain, largely focused around SDN. Xie et al. introduced techniques for statically analyzing the reachability properties of networks [107]. Some prominent recent domain specific verification engines include: Header Space Analysis [49] and NetPlumber [48], which introduced the idea of including location in the packet metadata, uniformizing packet transformations and topology and Veriflow [52], which functions as a real-time invariant

checker sitting between a controller and the network. Zhang and Malik [112] build a SAT-based verification framework that uses a similar network model to Header Space Analysis, but generalizes the model to allow uniform specification of correctness requirements[3]. Unlike the system described in Chapter 3, these tools work at several levels below the programmer, making it difficult to relate the results of verification back to the actual code. For example, if one of these verifiers says that a rule that was just inserted into the network breaks reachability, the programmer would have to determine why that rule was added by walking back through the controller that installed the rule, to the compiler that output the rules to be installed, and connect that back to the input policy, which itself was generated by another piece of code.

The model developed by Kazemian *et. al* [49] was the starting point for the network model in Chapter 5. Since their model only spoke of a single, static configuration, we extended the network semantics to include updates so we could model a network changing dynamically over time. In addition, while their model was used to help *describe* their algorithms, ours was used to help us state and prove various correctness properties of our system.

Finally, Foster *et al.* [25] present an equivalence checker for NetKAT based on very similar foundations as the one in this chapter. The system in this chapter was based on a rewrite of the code base in that paper, and shares a similar architecture and overall bisimulation algorithm. For more specific details on the difference between that paper and this one, see Section 3.6.

[3]Header Space Analysis required ad-hoc extensions of the model to support certain properties. For example, to detect forwarding loops, they extended the packet header to include a list of all visited ports

6.3.1 Network debugging

The NICE model checker [14] is a state-space exploration engine that can detect bugs in a full OpenFlow system: switches, controller, and control application. NICE searches for generic network bugs such as forwarding loops or black-holes, and allows the user to program application specific invariants to test. The system uses domain specific exploration strategies to reduce the complexity of the state space and find bugs more quickly. Portions of the Featherweight OpenFlow model in Chapter 4 are inspired by the bugs discovered in NICE.

Sethi *et al.* [90] extend the bounded verification approach of NICE to arbitrarily large numbers of packets with the use of application specific *non-interference* lemmas. The Kuai system [65] further scaled the model checking approach to much larger topologies, and automatically deals with unbounded sets of packets without the need for manual lemmas[4]

A number of projects have focused upon easing the difficulty of debugging network control and dataplanes by extending models of software debuggers to networks as in NDB [38]; automatically generating test packets to detect bugs in data planes [110]; or analyzing snapshots of network state with a SAT solver to detect failures in the dataplane [64].

6.3.2 Network verification

VeriCon [6] is a system that verifies the correctness of an SDN program for all topologies with a specific property, and all possible network inputs. Based upon Z3 [18], VeriCon specifies desired network behavior and acceptable topologies in first-order logic, rather than a networking specific language. Unlike most of the other systems here, they assume in-order,

[4]As an interesting coincidence, Kuai is built upon the PReach distributed model checker [12], which the author helped build in between undergrad and graduate school.

atomic installation of switch rules, an assumption that can be enforced in practice, but only with a high performance penalty.

The verification tool in Chapter 3 was originally based upon the NetKAT verifier in [25]. The differences between that tool and the one in this thesis are outlined in detail in the relevant chapter. Stewart [94] built a formal verification system for NetCore (a predecessor to NetKAT) based upon Hoare triples in Coq, and proved the verification tool itself correct with respect to a formal model of NetCore based upon the semantics in Chapter 4.

In a different vein, Dobrescu and Argyraki verify the correctness network dataplanes implemented in the Click software router framework [55]. They use symbolic execution to prove properties of software dataplanes that are routinely verified for hardware dataplanes such as crash-freedom or bounded execution.

6.4 Network updates

Updating networks without introducing undesired behavior has long been recognized as an important and difficult problem. Because of the great complexity of networks, even solutions that avoid problems in one protocol level can inadvertently introduce undesired behavior in another, nominally independent, protocol (see especially [99] for a surprising example of this interplay between IGP and BGP). Before the rise of SDN, most approaches focused upon manipulating the inputs to distributed routing protocols so that they would converge to desired configurations without transitioning through "bad states". A full overview of this area would be beyond the scope of this chapter, but a representative sampling can be found through these topics: performing network maintenance without disruption [91] [51] [28], handling topology changes [77] [92] [29], routing protocol migration [50] [102] [100], avoiding

disruptions during traffic engineering [26] [27], general techniques for avoiding introducing forwarding loops [30] [93], and general frameworks for updating distributed networking protocols [16] [84]. For a more thorough exploration of the area, see *e.g.* Vanbever's thesis [101].

Consensus Routing [45] seeks to eliminate transient errors, such as disconnectivity, that arise during BGP updates. In particular, Consensus Routing's "stable mode" is similar to our per-packet consistency, though computed in a distributed manner for BGP routes. On the other hand, Consensus Routing only applies to a single protocol (BGP), whereas our work may benefit any protocol or application developed in our framework. The BGP-LP mechanism from [46] is essentially per-packet consistency for BGP.

The dynamic software update problem is related to network update consistency. The problem of upgrading software in a general distributed system is addressed in [2]. The scope of that work differs in that the nodes being updated are general purpose computers, not switches, running general software.

The related problem of maintaining safety while updating firewall configurations has been addressed by Zhang *et. al* [111]. That work formalized a definition of safety similar in spirit to per-packet consistency, but limited to a single device.

6.4.1 Alternative abstractions

Since the original publication of the per-packet and per-flow consistency abstractions, there has been a large body of work proposing new abstractions and new mechanisms.

Customizable properties One such abstraction preserves specific, application specific properties through a general update specialization procedure. Instead preserving all possible *trace properties* as in per-packet consistency, a programmer can specify exactly the properties they want preserved, and achieve a faster update. The Dionysus system [44] dynamically schedules updates based upon the exact properties to be preserved, as well as the performance characteristics of the switches themselves[5]. However, Dionysus requires a rule dependency calculation that is specific to the general property being preserved, and thus can be automatically applied to different properties. The network synthesis approach [67] views network updates as a distributed programming problem and uses techniques from program synthesis to construct minimal update sequences that preserve desired network invariants specified in temporal logic. Similarly, the Customizeable Consistency Generator [115] analyses the desired invariant and synthesizes a update sequence that satisfies it, or resorts to the consistent update mechanisms outlined in Chapter 5 when no such sequence exists.

Alternative properties Consistent updates are guaranteed to preserve all *trace properties*, but not all network properties are *trace properties*. For example, guarantees about congestion, bandwidth or latency are not preserved by consistent updates. The Software-driven Wide Area Network (SWAN) [41] system used SDN to optimzie the network utilization of expensive WAN links. SWAN analyzed the problem of a *congestion-free* update: given a congestion-free network state with known loads and link capacities, compute a sequence of network updates that lead to target congestion-free state such that none of the intermediate states introduce congestion. They show that, in general, no such update may exist, but if $x\%$ of "scratch capacity" is left on each link, then an update sequence can be found of length at most $\lceil \frac{1}{x} \rceil - 1$. Similarly, the zUpdate [59] system provides an abstraction that guarantees

[5]Dionysus found that update times can vary dramatically between different switches. By dynamically adjusting its update schedule to the observed performance of the switches, they were able to achieve significant increases in update performance

a congestion-free, lossless network update.

Per-flow consistency generalizes per-packet consistency by preserving properties on related packets within a single flow. Inter-flow consistency [60] generalizes per-flow consistency by preserving constraints between different flows. For example, inter-flow consistency can guarantee that two specific flows are never colocated on the same link (*e.g.*for fault-tolerance), or that related flows in different directions (*e.g.*the two flows of a single TCP connection) are treated in a consistent manner.

6.4.2 Optimized update mechanisms

The update mechanisms described in this thesis may require high-switch rulespace overhead, or long convergence time. A number of optimizations that explore different tradeoffs in time and space have been proposed. If an update is split into several incremental updates, then the maximum rulespace overhead required can be greatly reduced [47]. Similarly, Luo *et al.* observe that by carefully exploiting OpenFlow wildcard rules, rules common to both the old and new configuration can be left on the switch [62]. This optimization is related to the subspace update outlined in Chapter 5, and if combined with that analysis can lead to highly compact updates. ESPRES [82] is an update scheduler, similar in spirit to Dionysus, that reorders, rate-limits, and prioritizes updates to avoid overloading the limited control plane bandwidth found in early generation OpenFlow switches. McGeer [69] describes an algorithm that completely removes the overhead of two-phase update by trading off the time required for the update and possibly introducing latency to packets traveling through the network during update.

The per-flow mechanisms described in this thesis assume that all packets in a single

flow arrive at the same ingress switch. Afek *et al.* describe a mechanism [1] that removes this restriction. Similarly, Zhao *et al.* [114] use an optimization approach to minimize the overhead and management required to preserve per-flow consistency using the flow binning scheme. Liu *et al.* [61] provide a heuristic algorithm to solve the per-flow optimization problem.

In a different and novel approach, Mizrahi and Moses [72] propose using precisely synchronized network clocks to schedule and perform network updates with minimal inconsistency windows.

CHAPTER 7

CONCLUSIONS

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." —Leslie Lamport

To someone who's never built or maintained one, a computer network must seem like the simplest of objects. You draw a network diagram, starting with the nodes you want to communicate, draw a graph fully connecting them, and then you go build it. And in fact, if the only function of your network is connectivity, it is not much more conceptually difficult than that[1]. Unfortunately, most computer networks have requirements beyond simple connectivity. Consider a pre-paid wireless network, such as the ones commonly found in airports or onboard airplanes. The network must provide full connectivity to paid subscribers, limited connectivity to unpaid users, enable new users to create accounts, track and bill traffic usage (but not traffic used for creating/checking accounts!), throttle or restrict traffic for users when their subscription ends, and protect users from one another. And it's not enough just to build the network: maintaining and operating it introduces a whole set of operational requirements such as active monitoring to detect failures, logging to debug connectivity problems, and fine-grained geographic tracking for analytics[2], just to name a few. All of these different requirements are traditionally implemented with their own set of protocols and mechanisms, many of which, by necessity, overlap and interact in complicated ways.

[1]Even in this case, things can quickly become complicated: which connected graph should be chosen? Should wired or wireless links be used? What happens when a link or node fails? How do you add new nodes to the network? What addressing scheme should you use? What nodes need to be able to broadcast to each other?

[2]Which access points are overutilized? Underutilized? How does utilization vary with time of day? *ad infinitum*

And yet, at its core, a network truly is a simple object. Almost all network functionality boils down to looking at a packet and deciding how to modify it and where to forward it. The techniques, languages, and tools presented in this thesis attack the complexity of networking by distilling it down to this simple core.

We began by describing a specification language that describes the desired flow of packets through a network at a high-level of abstraction, but still admits automatic verification of correct implementations. We then showed how to compile network programs into the low-level language of switches in a provably correct way that preserves the original verification promises. Finally, we showed how focusing on the behavior of a single packet through the network leads to a correctness criteria for network updates that provides strong reasoning guarantees about network behavior, even while it is in flux.

BIBLIOGRAPHY

[1] Yehuda Afek, Anat Bremler-Barr, and Liron Schiff. Ranges and cross-entrance consistency with openflow. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 233–234, New York, NY, USA, 2014. ACM.

[2] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 452–476, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] E. Al-Shaer and M.N. Alsaleh. Configchecker: A tool for comprehensive security configuration analytics. In *Configuration Analytics and Automation (SAFECONFIG), 2011 4th Symposium on*, pages 1–2, Oct 2011.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[5] Andrew W. Appel. Verified software toolchain. In *ESOP*, 2011.

[6] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Notices*, volume 49, pages 282–293. ACM, 2014.

[7] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: a novel firewall management toolkit. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 17–31, 1999.

[8] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 328–341, New York, NY, USA, 2016. ACM.

[9] Yves Bertot and Pierre Casteran. Interactive theorem proving and program development: Coq'Art the calculus of inductive constructions. In *EATCS Texts in Theoretical Computer Science*. Springer-Verlag, 2004.

[10] W. R. Bevier. Kit: A study in operating system verification. *IEEE Trans. Softw. Eng.*, 15(11):1382–1396, Nov 1989.

[11] William R Bevier, Warren A Hunt Jr, J Strother Moore, and William D Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.

[12] Brad Bingham, Jesse Bingham, Flavio M. de Paula, John Erickson, Gaurav Singh, and Mark Reitblatt. Industrial strength distributed explicit state model checking. In *Proceedings of the 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology*, PDMC-HIBI '10, pages 28–36, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Steve Bishop, Matthew Fairbairn, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Rigorous specification and validation for tcp/ip and the sockets api.

[14] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[15] Martin Casado, Nate Foster, and Arjun Guha. Abstractions for software-defined networks. *Commun. ACM*, 57(10):86–95, Sep 2014.

[16] Xu Chen, Z. Morley Mao, and Jacobus Van der Merwe. Pacman: A platform for automated and controlled network operations and configuration management. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 277–288, New York, NY, USA, 2009. ACM.

[17] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, 2011.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] David Erickson et al. A demonstration of virtual machine mobility in an OpenFlow network, Aug 2008. Demo at *ACM SIGCOMM*.

[20] Nick Feamster and Hari Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, May 2005.

[21] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proc. ACM SIGCOMM '13*, Hong Kong, China, August 2013.

[22] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Hierarchical policies for software defined networks. In *HotSDN*, 2012.

[23] N. Foster, A. Guha, M. Reitblatt, A. Story, M.J. Freedman, N.P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-defined networks. *Communications Magazine, IEEE*, 51(2):128–134, February 2013.

[24] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, Sep 2011.

[25] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for netkat. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 343–355, New York, NY, USA, 2015. ACM.

[26] P. Francois and O. Bonaventure. Avoiding transient loops during IGP convergence in ip networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, pages 237–247 vol. 1, March 2005.

[27] Pierre Francois and Olivier Bonaventure. Avoiding transient loops during the convergence of link-state routing protocols. *IEEE/ACM Trans. on Networking*, Dec 2007.

[28] Pierre Francois, Pierre-Alain Coste, Bruno Decraene, and Olivier Bonaventure. Avoiding disruptions during maintenance operations on BGP sessions. *IEEE Trans. on Network and Service Management*, Dec 2007.

[29] Pierre Francois, Mike Shand, and Olivier Bonaventure. Disruption-free topology reconfiguration in OSPF networks. In *IEEE INFOCOM*, May 2007.

[30] Jing Fu, P. Sjodin, and G. Karlsson. Loop-free updates of forwarding tables. *Network and Service Management, IEEE Transactions on*, 5(1):22–35, March 2008.

[31] Wouter Gelade and Frank Neven. Succinctness of the Complement and Intersection of Regular Expressions. In Susanne Albers and Pascal Weil, editors, *25th International*

Symposium on Theoretical Aspects of Computer Science, volume 1 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 325–336, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[32] D. I. Good and B. A. Wichmann. Mechanical proofs about computer programs [and discussion]. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 312(1522):389–409, 1984.

[33] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA, 1993.

[34] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[35] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-Verified Network Controllers . In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Seattle, WA*, Jun 2013.

[36] J.D. Guttman. Filtering postures: local enforcement for global policies. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 120–129, May 1997.

[37] Nikhil Handigol, Brandon Heller, Vimal Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, 2012.

[38] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 55–60. ACM, 2012.

[39] Nikhil Handigol, Srinivasan Seetharaman, Mario Flajslik, Nick McKeown, and Ramesh Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow, Aug 2009. Demo at *ACM SIGCOMM*.

[40] Tony Hoare. The verifying compiler: A grand challenge for computing research. *JACM*, 50(1):63–69, Jan 2003.

[41] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan.

In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 15–26, New York, NY, USA, 2013. ACM.

[42] Warren A Hunt Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[43] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[44] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 539–550, New York, NY, USA, 2014. ACM.

[45] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus routing: The Internet as a distributed system. In *NSDI*, Apr 2008.

[46] Dina Katabi, Nate Kushman, and John Wrocklawski. A Consistency Management Layer for Inter-Domain Routing. Technical Report MIT-CSAIL-TR-2006-006, Cambridge, MA, Jan 2006.

[47] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 49–54, New York, NY, USA, 2013. ACM.

[48] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking using Header Space Analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr 2013.

[49] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[50] Eric Keller, Jennifer Rexford, and Jacobus Van Der Merwe. Seamless BGP migration with router grafting. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 16–16, Berkeley, CA, USA, 2010. USENIX Association.

[51] Ram Keralapura, Chen-Nee Chuah, and Yueyue Fan. Optimal strategy for graceful

network upgrade. In *Proceedings of the 2006 SIGCOMM Workshop on Internet Network Management*, INM '06, pages 83–88, New York, NY, USA, 2006. ACM.

[52] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Lombard, IL*, Apr 2013.

[53] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, May 2015. USENIX Association.

[54] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an OS kernel. In *SOSP*, 2009.

[55] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug 2000.

[56] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 14–33. 1996.

[57] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[58] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, Jul 2009.

[59] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 411–422, New York, NY, USA, 2013. ACM.

[60] Weijie Liu, Rakesh B Bobba, Sibin Mohan, and Roy H Campbell. Inter-flow consistency: Novel SDN update abstraction for supporting inter-flow constraints. In *Proceedings of the Network and Distributed System Security Symposium*, Network and Distributed System Security Symposium 2015, 2015.

[61] Yujie Liu, Yong Li, Yue Wang, Athanasios V. Vasilakos, and Jian Yuan. Achieving efficient and fast update for multiple flows in software-defined networks. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, DCC '14, pages 77–82, New York, NY, USA, 2014. ACM.

[62] Shouxi Luo, Hongfang Yu, and Lemin Li. Consistency is not easy: How to use two-phase update for wildcard rules? *Communications Letters, IEEE*, 19(3):347–350, March 2015.

[63] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *ASPLOS*, 2013.

[64] Haohui Mai, Ahmed Khurshid, Raghit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[65] Rwitajit Majumdar, Sai Deep Tetali, and Zhen Wang. Kuai: A model checker for software-defined networks. In *Formal Methods in Computer-Aided Design (FMCAD), 2014*, pages 163–170. IEEE, 2014.

[66] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Towards a verified relational database management system. In *POPL*, 2010.

[67] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient synthesis of network updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 196–207, New York, NY, USA, 2015. ACM.

[68] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *ICFP*, 2010.

[69] Rick McGeer. A correct, zero-overhead protocol for network updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 161–162, New York, NY, USA, 2013. ACM.

[70] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[71] Saber Mirzaei, Sanaz Bahargam, Richard Skowyra, Assaf Kfoury, and Azer Bestavros.

Using Alloy to formally model and reason about an openflow network switch. *Computer Science Department, Boston University, Tech. Rep*, 7, 2013.

[72] Tal Mizrahi and Yoram Moses. Time-based updates in software defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 163–164, New York, NY, USA, 2013. ACM.

[73] J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. Curtis, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *SIGCOMM*, Aug 2011.

[74] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, 2012.

[75] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[76] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *PLDI*, 2012.

[77] J Moy, Padma Pillay-Esnault, and Acee Lindem. Graceful OSPF restart. Technical report, 2003. RFC 3623.

[78] Tim Nelson, Andrew D. Ferguson, Michael J.G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 519–531, Seattle, WA, 2014. USENIX Association.

[79] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 79–84, New York, NY, USA, 2013. ACM.

[80] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[81] Sam Owre, Sreeranga Rajan, John M Rushby, Natarajan Shankar, and Mandayam Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer Aided Verification*, pages 411–414. Springer, 1996.

[82] Peter Perešíni, Maciej Kuzniar, Marco Canini, and Dejan Kostić. Espres: Easy scheduling and prioritization for SDN. In *Presented as part of the Open Networking Summit 2014 (ONS 2014)*, Santa Clara, CA, 2014. USENIX.

[83] Junaid Qadir and Osman Hasan. Applying formal methods to networking: Theory, techniques, and applications. *Communications Surveys & Tutorials, IEEE*, 17(1):256–291, 2015.

[84] S. Raza, Y. Zhu, and C-N. Chuah. Graceful network operations. In *IEEE INFOCOM*, Apr 2009.

[85] S. Raza, Y. Zhu, and C-N. Chuah. Graceful network state migrations. *IEEE/ACM Trans. on Networking*, 19(4), Aug 2011.

[86] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 109–114, New York, NY, USA, 2013. ACM.

[87] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Aug 2012.

[88] Frank La Rue. Report of the Special Rapporteur on the promotion and protection of the right to freedom of opinion and expression. Technical report, United Nations, Human Rights Council, 05 2011.

[89] J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, Oct 2000.

[90] Divjyot Sethi, Srinivas Narayana, and Sharad Malik. Abstractions for model checking SDN controllers. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 145–148. IEEE, 2013.

[91] A. Shaikh, R. Dube, and A. Varma. Avoiding instability during graceful shutdown of multiple OSPF routers. *Networking, IEEE/ACM Transactions on*, 14(3):532–542, June 2006.

[92] Mike Shand and Les Ginsberg. Restart signaling for IS-IS. Technical report, 2008. RFC 5306.

[93] Lei Shi, Jing Fu, and Xiaoming Fu. Loop-free forwarding table updates with minimal link overflow. In *Proceedings of the 2009 IEEE International Conference on Communications*, ICC'09, pages 2091–2096, Piscataway, NJ, USA, 2009. IEEE Press.

[94] Gordon Stewart. Computational verification of network programs in Coq. In *Certified Programs and Proofs*, pages 33–49. Springer, 2013.

[95] LJ Stockmeyer. *The complexity of decision procedures in Automata Theory and Logic*. PhD thesis, PhD thesis, MIT, Project MAC Technical Report TR-133, 1974.

[96] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *POPL*, 2012.

[97] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94, Oct 2007.

[98] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martín Casado, and Rob Sherwood. On controller performance in software-defined networks. In *HotICE*, 2012.

[99] L. Vanbever, S. Vissicchio, L. Cittadini, and O. Bonaventure. When the cure is worse than the disease: The impact of graceful IGP operations on BGP. In *INFOCOM, 2013 Proceedings IEEE*, pages 2220–2228, April 2013.

[100] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Lossless migrations of link-state IGPs. *Networking, IEEE/ACM Transactions on*, 20(6):1842–1855, Dec 2012.

[101] Laurent Vanbever. *Methods and Techniques for Disruption-Free Network Reconfiguration*. PhD thesis, University of Louvain, 2012.

[102] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. Seamless network-wide IGP migration. In *SIGCOMM*, Aug 2011.

[103] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, Jan 2011.

[104] Anduo Wang, Limin Jia, Changbin Lio, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally verifiable networking. In *HotNets*, 2009.

[105] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE*, Mar 2011.

[106] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. *PLDI. ACM*, 2015.

[107] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.

[108] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[109] W. Young. Verified compilation in micro-Gypsy. In *TAV*, 1989.

[110] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *CoNEXT*, 2012.

[111] Charles C. Zhang, Marianne Winslett, and Carl A. Gunter. On the safety and efficiency of firewall policy deployment. In *IEEE Symp. on Security and Privacy*, 2007.

[112] Shuyuan Zhang and Sharad Malik. Sat based verification of network data planes. In *Automated Technology for Verification and Analysis*, pages 496–505. Springer, 2013.

[113] Jianzho Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, 2012.

[114] Kun Zhao, Qing Li, and Yong Jiang. Flow-level consistent update in SDN based on k-prefix covering. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 1884–1889, Dec 2014.

[115] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. Enforcing customizable consistency properties in software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2015.

APPENDIX A

PROOFS

A.1 Proofs for Chapter 3

We first prove a lemma characterizing the translation of path expressions:

Lemma 5 (Equivalence of Path expression translation)**.** *For every path expression P, $G(P)$ $= G(\llparenthesis P \rrparenthesis)$*

Proof. Proof by structural induction on the Pathetic path expression policy P.

Case ϵ

$$\begin{aligned} G(\epsilon) &= \{\alpha{\cdot}\pi_\alpha \mid \alpha \in \mathsf{At}\} \\ &= G(1) \\ &= G(\llparenthesis \epsilon \rrparenthesis) \end{aligned}$$

Case $\emptyset$

$$\begin{aligned} G(\emptyset) &= \emptyset \\ &= G(0) \\ &= G(\llparenthesis \epsilon \rrparenthesis) \end{aligned}$$

Case S

$$\begin{aligned} G(S) &= \{\alpha{\cdot}\pi{\cdot}\mathsf{dup}\pi \mid \alpha \in \mathsf{At}, \pi = \pi_\alpha[S/sw]\} \\ &= G(sw \leftarrow S \cdot \mathsf{dup}) \\ &= G(\llparenthesis S \rrparenthesis) \end{aligned}$$

Case $\star$

$$\begin{aligned} G(\star) &= \{\alpha{\cdot}\pi{\cdot}\mathsf{dup}\pi \mid \alpha \in \mathsf{At}, \pi = \pi_\alpha[S/sw], S \in \mathtt{Sw}\} \\ &= G(\sum_{S \in \mathtt{Sw}} sw \leftarrow S \cdot \mathsf{dup}) \\ &= G((\!|\star|\!)) \end{aligned}$$

Case $\overline{P}$

$$\begin{aligned} G(\overline{P}) &= \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \setminus G(P) \\ &= \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \setminus G((\!|P|\!)) && \text{By the induction hypothesis} \\ &= G(\overline{(\!|P|\!)}) \\ &= G((\!|\overline{P}|\!)) \end{aligned}$$

Case $P.P'$

$$\begin{aligned} G(P.P') &= G(P) \diamond G(P') \\ &= G((\!|P|\!)) \diamond G((\!|P'|\!)) && \text{By the induction hypothesis} \\ &= G((\!|P|\!) \cdot (\!|P'|\!)) \\ &= G((\!|P.P'|\!)) \end{aligned}$$

Case $P|P'$

$$\begin{aligned} G(P|P') &= G(P) \cup G(P') \\ &= G((\!|P|\!)) \cup G((\!|P'|\!)) && \text{By the induction hypothesis} \\ &= G((\!|P|\!) + (\!|P'|\!)) \\ &= G((\!|P|P'|\!)) \end{aligned}$$

Case $P \cap P'$

$$G(P \cap P') = G(P) \cap G(P')$$

$$
\begin{aligned}
&= G(\llparenthesis P \rrparenthesis) \cap G(\llparenthesis P' \rrparenthesis) && \text{By the induction hypothesis} \\
&= G(\llparenthesis P \rrparenthesis \cap \llparenthesis P' \rrparenthesis) \\
&= G(\llparenthesis P \cap P' \rrparenthesis)
\end{aligned}
$$

Case P^*

$$
\begin{aligned}
G(P^*) &= \bigcup_{i \geq 0} G(P)^i \\
&= \bigcup_{i \geq 0} G(\llparenthesis P \rrparenthesis)^i && \text{By the induction hypothesis} \\
&= G(\llparenthesis P \rrparenthesis^*) \\
&= G(\llparenthesis P^* \rrparenthesis)
\end{aligned}
$$

□

Theorem 13 (Theorem 1)**.** *For every Pathetic program* ϕ*,* $G(\phi) = G(\llparenthesis \phi \rrparenthesis)$

Proof. Proof by induction on ϕ.

Case $a \Rightarrow P$

$$
\begin{aligned}
G(a \Rightarrow P) &= G(a) \diamond G(P) \\
&= G(a) \diamond G(\llparenthesis P \rrparenthesis) && \text{By Lemma 5} \\
&= G(a \cdot \llparenthesis P \rrparenthesis) \\
&= G(\llparenthesis a \Rightarrow P \rrparenthesis)
\end{aligned}
$$

Case $\phi \uplus \phi'$ Immediate by induction and definition of $\llparenthesis \phi \uplus \phi' \rrparenthesis$.

Case $\phi \pitchfork \phi'$ Immediate by induction and definition of $\llparenthesis \phi \pitchfork \phi' \rrparenthesis$.

□

Theorem 14 (Theorem 2)**.** $p \vDash \phi$ *iff* $(\!|\phi|\!) \cap \overline{p} \equiv \mathsf{0}$.

Proof.

$$\begin{aligned}
p \vDash \phi &\iff G(p) \subseteq G(\phi) \\
&\iff \overline{G(p)} \cap G(\phi) = \emptyset \\
&\iff G(\overline{p}) \cap G(\phi) = \emptyset \\
&\iff G(\overline{p}) \cap G((\!|\phi|\!)) = \emptyset \\
&\iff G(\overline{p} \cap G((\!|\phi|\!))) = \emptyset \\
&\iff G(\overline{p} \cap G((\!|\phi|\!))) = G(\mathsf{0}) \\
&\iff \overline{p} \cap G((\!|\phi|\!)) \equiv \mathsf{0}
\end{aligned}$$

□

Corollary 3 (Corollary 1)**.** $p \vDash \phi$ *iff* $p \leq (\!|\phi|\!)$.

Theorem 15 (NetKAT$(-, \cap)$ is a conservative extension of NetKAT)**.** *For every complement and intersection-free policy* p, $[\![p]\!] = [\![p]\!]_{NK}$, *where* $[\![\cdot]\!]_{NK}$ *is the original NetKAT semantics.*

Proof. Immediate by induction upon p. □

Theorem 16 (Theorem 3)**.** *For all dup-free policies* p *and* q, *if* $p \equiv q$ *is provable by the NetKAT*$(-, \cap)$ *axioms, then* $[\![p]\!]_{-\mathsf{dup}} = [\![q]\!]_{-\mathsf{dup}}$.

Proof. Proof by structural induction on the derivation of $p \equiv q$ with a case analysis on the last rule used. Soundness of the NetKAT axioms follows from Theorem 15.

The soundness of the axioms INTER-*, PAR-INTER-DIST, COMP-PAR, and COMP-INTER follow trivially from the semantics and basic properties of union/intersection.

INTER-MOD-DIST-LEFT

$$\begin{aligned}
[\![f \leftarrow n \cdot (p \cap q)]\!]_{-\mathsf{dup}}\ pk &= ([\![f \leftarrow n]\!]_{-\mathsf{dup}} \bullet [\![p \cap q]\!]_{-\mathsf{dup}})\ pk \\
&= \bigcup_{pk' \in [\![f \leftarrow n]\!]_{-\mathsf{dup}}\ pk} [\![p \cap q]\!]_{-\mathsf{dup}}\ pk' \\
&= \bigcup_{pk' \in \{pk[n/f]\}} [\![p \cap q]\!]_{-\mathsf{dup}}\ pk' \\
&= [\![p \cap q]\!]_{-\mathsf{dup}}\ pk[n/f] \\
&= [\![p]\!]_{-\mathsf{dup}}\ pk[n/f] \cap [\![q]\!]_{-\mathsf{dup}}\ pk[n/f] \\
&= [\![f \leftarrow n \cdot p]\!]_{-\mathsf{dup}}\ pk \cap [\![f \leftarrow n \cdot q]\!]_{-\mathsf{dup}}\ pk \\
&= [\![(f \leftarrow n \cdot p) \cap (f \leftarrow n \cdot q)]\!]_{-\mathsf{dup}}\ pk
\end{aligned}$$

COMP-FILTER

$$\begin{aligned}
[\![\overline{f = n}]\!]_{-\mathsf{dup}}\ pk &= \mathcal{PK} \setminus [\![f = n]\!]_{-\mathsf{dup}}\ pk \\
&= \mathcal{PK} \setminus \{pk \mid pk[f] = n\}
\end{aligned}$$

Reasoning by cases:

If $pk(f) \neq n$

$[\![\neg f = n \cdot \sum_\pi \pi]\!]\ pk = \mathcal{PK}$

If $pk(f) = n$

$[\![\neg f = n \cdot \sum_\pi \pi]\!]\ pk = \emptyset$ and $[\![\sum_\alpha \sum_{\pi \neq \pi_\alpha} \alpha \cdot \pi]\!] = \mathcal{PK} \setminus \{pk\}$.

COMP-MOD

$$\begin{aligned}
[\![\sum_{\alpha} \sum_{\pi \neq \pi_{\alpha[f \leftarrow n]}} \alpha \cdot \pi]\!]\ pk &= \bigcup_{\alpha, \pi \neq \pi_{\alpha[f \leftarrow n]}} [\![\alpha \cdot \pi]\!]\ pk \\
&= \bigcup_{\pi \neq \pi_{\alpha[f \leftarrow n]}} [\![\pi]\!]\ pk \qquad \text{For the unique } \alpha \text{ such that } \alpha(\pi) \\
&= \mathcal{PK} \setminus \{pk[f/n]\} \\
&= [\![\overline{f \leftarrow n}]\!]\ pk
\end{aligned}$$

COMP-SEQ

$$\begin{aligned}
[\![\overline{p \cdot q}]\!]_{-\mathsf{dup}}\ pk &= \overline{[\![p \cdot q]\!]_{-\mathsf{dup}}\ pk} \\
&= \overline{\bigcup_{pk' \in [\![p]\!]_{-\mathsf{dup}}\ pk} [\![q]\!]_{-\mathsf{dup}}\ pk'} \\
&= \bigcap_{pk' \in [\![p]\!]_{-\mathsf{dup}}\ pk} \overline{[\![q]\!]_{-\mathsf{dup}}\ pk'} \\
&= \bigcap_{pk' \in [\![p]\!]_{-\mathsf{dup}}\ pk} [\![\overline{q}]\!]_{-\mathsf{dup}}\ pk' \\
&= \bigcap_{pk'} ([pk' \notin [\![p]\!]_{-\mathsf{dup}}\ pk] \cdot \mathsf{all} \cup [\![\overline{q}]\!]_{-\mathsf{dup}}\ pk') \\
&= \bigcap_{pk'} \left([pk' \in \overline{[\![p]\!]_{-\mathsf{dup}}\ pk}] \cdot \mathsf{all} \cup [\![\overline{q}]\!]_{-\mathsf{dup}}\ pk' \right) \\
&= \bigcap_{pk'} ([pk' \in [\![\overline{p}]\!]_{-\mathsf{dup}}\ pk] \cdot \mathsf{all} \cup [\![\overline{q}]\!]_{-\mathsf{dup}}\ pk') \\
&= \bigcap_{pk'} ([\![\overline{p} \cdot \alpha_{pk'} \cdot \mathsf{all}]\!]_{-\mathsf{dup}}\ pk \cup [\![\overline{q}]\!]_{-\mathsf{dup}}\ pk') \\
&= \bigcap_{pk'} ([\![\overline{p} \cdot \alpha_{pk'} \cdot \mathsf{all}]\!]_{-\mathsf{dup}}\ pk \cup [\![\pi_{pk'} \cdot \overline{q}]\!]_{-\mathsf{dup}}\ pk) \\
&= \bigcap_{pk'} ([\![\overline{p} \cdot \alpha_{pk'} \cdot \mathsf{all} + \pi_{pk'} \cdot \overline{q}]\!]_{-\mathsf{dup}}\ pk) \\
&= [\![\bigcap_{pk'} \overline{p} \cdot \alpha_{pk'} \cdot \mathsf{all} + \pi_{pk'} \cdot \overline{q}]\!]_{-\mathsf{dup}}\ pk
\end{aligned}$$

□

Theorem 17 (Theorem 4)**.** *For all policies p and q, if*

$$p \equiv q$$

in the equational theory generated by the NetKAT$(-, \cap)$ axioms minus COMP-FILTER*,* COMP-MOD*, and* COMP-SEQ*, then*

$$[\![p]\!] = [\![q]\!]$$

Proof. Proof by structural induction on the derivation of $p \equiv q$ with a case analysis on the last rule used. Soundness of the NetKAT axioms follows from Theorem 15.

The soundness of the axioms INTER-* and PAR-INTER-DIST follow trivially from the semantics and basic properties of union/intersection. This leaves only the axiom INTER-MOD-DIST-LEFT.

$$\begin{aligned}
[\![f \leftarrow n \cdot (p \cap q)]\!]\ pk{::}h &= ([\![f \leftarrow n]\!] \bullet [\![p \cap q]\!])\ pk{::}h \\
&= \bigcup_{pk'::h' \in [\![f \leftarrow n]\!]\ pk::h} [\![p \cap q]\!]\ pk'::h' \\
&= \bigcup_{pk'::h' \in \{pk[n/f]::h\}} [\![p \cap q]\!]\ pk'::h' \\
&= [\![p \cap q]\!]\ pk[n/f]::h \\
&= [\![p]\!]\ pk[n/f]::h \cap [\![q]\!]\ pk[n/f]::h \\
&= [\![f \leftarrow n \cdot p]\!]\ pk::h \cap [\![f \leftarrow n \cdot q]\!]\ pk::h
\end{aligned}$$

$$= [\![(f \leftarrow n \cdot p) \cap (f \leftarrow n \cdot q)]\!]\ pk{::}h$$

□

A.1.1 Completeness

To prove completeness of the dup-free NetKAT$(-, \cap)$ axioms, we show that every dup-free NetKAT$(-, \cap)$ term is provably equivalent to a dup-free NetKAT term, and then appeal to the completeness of the NetKAT axioms. To make the induction work, we actually show a stronger theorem: every dup-free NetKAT$(-, \cap)$ term is provably equivalent to a dup-free,*-free reduced NetKAT term.

Lemma 6. $(\sum_i \alpha_i \cdot \pi_i) \cap (\sum_j \alpha'_j \cdot \pi'_j) \equiv \sum_k \alpha_k \cdot \pi_k$ *such that* $\forall k \exists i, j$ *s.t.* $\alpha_k = \alpha_i = \alpha'_j$ *and* $\pi_k = \pi_i = \pi'_j$.

Lemma 7. $\overline{\alpha \cdot \pi} \equiv (\sum_{\alpha' \neq \alpha} \sum_{\pi'} \alpha' \cdot \pi') + \sum'_{\alpha} \sum_{\pi' \neq \pi} \alpha' \cdot \pi'$ *is provable.*

Lemma 8. *Every* dup*-free NetKAT$(-, \cap)$ policy is provably equivalent to a sum of reduced* **-free,*dup*-free NetKAT policies.*

Proof. Proof by structural induction upon the policy p, with a case analysis on the last syntax rule.

Because the NetKAT$(-, \cap)$ axioms are a conservative extension of the NetKAT axioms, all cases except $p \cap q$,$\overline{p}$, and p^* follow immediately from the induction hypothesis and fact that every dup-free NetKAT policy is provably equivalent to a sum of dup-free reduced NetKAT policies (Lemma 9 in [4]).

Case $p \cap q$ By the induction hypothesis, $p \equiv \sum_i \alpha_i \cdot \pi_i$, and $q \equiv \sum_j \alpha'_j \cdot \pi'_j$. By Lemma 6, this is provably equal to a term $\sum_k \alpha_k \cdot \pi_k$, which is in the form desired.

Case $\overline{p}$ By the induction hypothesis, $p \equiv \sum_i \alpha_i \cdot \pi_i$.

$$\begin{aligned}
\overline{p} &\equiv \overline{\sum_i \alpha_i \cdot \pi_i} \\
&\equiv \prod_i \overline{\alpha_i \cdot \pi_i} && \text{By COMP-PAR} \\
&\equiv \prod_i \left(\sum_j \alpha_{i,j} \cdot \pi_{i,j} \right) && \text{By Lemma 7} \\
&\equiv \sum_k \alpha'_k \cdot \pi'_k && \text{For some } \alpha'_k, \pi'_k \text{ by INTER-PAR-DIST and induction}
\end{aligned}$$

Case p^* By the induction hypothesis, $p \equiv \sum_i \alpha_i \cdot \pi_i$. Thus, $p^* \equiv (\sum_i \alpha_i \cdot \pi_i)^*$, which is a dup-free NetKAT term. By a theorem of NetKAT (Lemma 9 in [4]), every dup-free NetKAT term is provably equivalent to a dup-free, *-free reduced NetKAT term.

□

Theorem 18 (Theorem 5)**.** *The axioms for NetKAT($-, \cap$) shown in Figure 3.9 plus the NetKAT axioms (minus* PA-DUP-FILTER-COMM*) are complete for the dup-free fragment.*

Proof. Follows directly from Lemma 8 and the proof of NetKAT completeness (Theorem 2 in [4]). □

A.1.2 NetKAT($-, \cap$) Derivatives

Lemma 9 ()**.** $E_{\alpha,\beta}(p) \equiv E(p)(\alpha)(\beta)$

Proof. Proof by structural induction upon the term p.

Case π

$$\begin{aligned} E_{\alpha,\beta}(\pi) &= [\pi = \pi_\beta] \\ &\equiv [\pi \circ \alpha = \pi_\beta] \\ &\equiv E_\pi(\alpha)(\beta) \\ &\equiv E(\pi)(\alpha)(\beta) \end{aligned}$$

Case b

$$\begin{aligned} E_{\alpha,\beta}(b) &= [\alpha = \beta \leq b] \\ &\equiv E_b(\alpha)(\beta) \\ &\equiv E(b)(\alpha)(\beta) \end{aligned}$$

Case $p + q$

$$\begin{aligned} E_{\alpha,\beta}(p+q) &= E_{\alpha,\beta}(p) + E_{\alpha,\beta}(q) \\ &\equiv E(p)(\alpha)(\beta) + E(q)(\alpha)(\beta) \\ &\equiv E(p) + E(q)(\alpha)(\beta) \\ &\equiv E(p+q)(\alpha)(\beta) \end{aligned}$$

Case $p \cap q$

$$\begin{aligned} E_{\alpha,\beta}(p \cap q) &= E_{\alpha,\beta}(p) \cdot E_{\alpha,\beta}(q) \\ &\equiv E(p)(\alpha)(\beta) \cdot E(q)(\alpha)(\beta) \\ &\equiv E(p) \cap E(q)(\alpha)(\beta) \\ &\equiv E(p \cap q)(\alpha)(\beta) \end{aligned}$$

Case $p \cdot q$

$$\begin{aligned} E_{\alpha,\beta}(p \cdot q) &= \sum_{\gamma} E_{\alpha,\gamma}(p) \cdot E_{\gamma,\beta}(q) \\ &\equiv \sum_{\gamma} E(p)(\alpha)(\gamma) \cdot E(q)(\gamma)(\beta) \\ &\equiv E(p) \cdot E(q)(\alpha)(\beta) \\ &\equiv E(p \cdot q)(\alpha)(\beta) \end{aligned}$$

Case $\overline{p}$

$$\begin{aligned} E_{\alpha,\beta}(\overline{p}) &= \overline{E_{\alpha,\beta}(p)} \\ &\equiv \overline{E(p)(\alpha)(\beta)} \\ &\equiv \overline{E(p)}(\alpha)(\beta) \\ &\equiv E(\overline{p})(\alpha)(\beta) \end{aligned}$$

Case p^*

$$\begin{aligned} E_{\alpha,\beta}(p^*) &= [\alpha = \beta] + \sum_{\gamma} E_{\alpha,\gamma}(p) \cdot E_{\gamma,\beta}(p^*) \\ &\equiv [\alpha = \beta] + \sum_{\gamma} E(p)(\alpha)(\gamma) \cdot E(p^*)(\gamma)(\beta) \\ &\equiv E(p^*)(\alpha)(\beta) \end{aligned}$$

□

Lemma 10.

$$D'_{\alpha,\beta}(p) \equiv \sum_{(e,d)\in D(p)} [e(\alpha)(\beta)] \cdot d$$

Proof. Proof by structural induction upon the term p.

Case $p = f \leftarrow n$ In this case, $D(p)$ is the empty set, which is equivalent to $D'_{\alpha\beta}(f \leftarrow n) = 0$.

Case $p = a$ In this case, $D(p)$ is the empty set, which is equivalent to $D'_{\alpha\beta}(a) = 0$.

Case $p = \mathsf{dup}$

$$\begin{aligned} D'_{\alpha\beta}(\mathsf{dup}) &= [\alpha = \beta] \\ &\equiv E_{\mathsf{1}}(\alpha)(\beta) \\ &\equiv E_{\mathsf{1}}(\alpha)(\beta) \cdot \mathsf{1} \\ &\equiv \sum_{(e,d)\in\{(E(\mathsf{1}),\mathsf{1})\}} [e(\alpha)(\beta)] \cdot d \\ &\equiv \sum_{(e,d)\in D(\mathsf{dup})} [e(\alpha)(\beta)] \cdot d \end{aligned}$$

Case $p = e_1 + e_2$

$$\begin{aligned} D'_{\alpha\beta}(e_1 + e_2) &= D'_{\alpha\beta}(e_1) + D'_{\alpha\beta}(e_2) \\ &\equiv \left(\sum_{(e,d)\in D(e_1)} [e(\alpha)(\beta)] \cdot d \right) + \left(\sum_{(e,d)\in D(e_2)} [e(\alpha)(\beta)] \cdot d \right) \quad \text{By induction} \\ &\equiv \sum_{(e,d)\in D(e_1)\cup D(e_2)} [e(\alpha)(\beta)] \cdot d \\ &\equiv \sum_{(e,d)\in D(p)} [e(\alpha)(\beta)] \cdot d \end{aligned}$$

Case $p = q \cdot r$

$$D'_{\alpha\beta}(e_1 \cdot e_2) = D'_{\alpha\beta}(e_1) \cdot e_2 + \sum_{\gamma} E_{\alpha\gamma}(e_1) \cdot D'_{\gamma\beta}(e_2)$$

$$\equiv \left(\sum_{(e,d)\in D(e_1)} [e(\alpha)(\beta)] \cdot d\right) \cdot e_2 + \sum_\gamma E_{\alpha\gamma}(e_1) \cdot \left(\sum_{(e',d')\in D(e_2)} [e'(\gamma)(\beta)] \cdot d'\right)$$

By induction

$$\equiv \left(\sum_{(e,d)\in D(e_1)} [e(\alpha)(\beta)] \cdot d\right) \cdot e_2 + \sum_\gamma E(e_1)(\alpha)(\gamma) \cdot \left(\sum_{(e',d')\in D(e_2)} [e'(\gamma)(\beta)] \cdot d'\right)$$

By Lemma 9

$$\equiv \left(\sum_{(e,d)\in D(e_1)} [e(\alpha)(\beta)] \cdot d\right) \cdot e_2 + \sum_\gamma \left(\sum_{(e',d')\in D(e_2)} E(e_1)(\alpha)(\gamma) \cdot [e'(\gamma)(\beta)] \cdot d'\right)$$

$$\equiv \left(\sum_{(e,d)\in D(e_1)} [e(\alpha)(\beta)] \cdot d\right) \cdot e_2 + \left(\sum_{(e',d')\in D(e_2)} \sum_\gamma E(e_1)(\alpha)(\gamma) \cdot [e'(\gamma)(\beta)] \cdot d'\right)$$

$$\equiv \left(\sum_{(e,d)\in D(e_1)} [e(\alpha)(\beta)] \cdot d\right) \cdot e_2 + \left(\sum_{(e',d')\in D(e_2)} [(E(e_1)\cdot e')(\alpha)(\beta)] \cdot d'\right)$$

$$\equiv \left(\sum_{(e,d)\in D(e_1)\cdot e_2} [e(\alpha)(\beta)] \cdot d\right) + \left(\sum_{(e',d')\in E(e_1)\cdot D(e_2)} [e'(\alpha)(\beta)] \cdot d'\right)$$

$$\equiv \sum_{(e,d)\in D(e_1)\cdot e_2 \cup E(e_1)\cdot D(e_2)} [e(\alpha)(\beta)] \cdot d$$

$$\equiv \sum_{(e,d)\in D(p)} [e(\alpha)(\beta)] \cdot d$$

Case $p = q \cap r$

$$D'_{\alpha\beta}(q \cap r) = D'_{\alpha\beta}(q) \cap D'_{\alpha\beta}(r)$$

$$\equiv \left(\sum_{(e_1,d_1)\in D(q)} [e_1(\alpha)(\beta)] \cdot d_1\right) \cap \left(\sum_{(e_2,d_2)\in D(r)} [e_2(\alpha)(\beta)] \cdot d_2\right)$$

By induction

$$\equiv \sum_{(e_1,d_1)\in D(q),(e_2,d_2)\in D(r)} ([e_1(\alpha)(\beta)] \cdot d_1) \cap ([e_2(\alpha)(\beta)] \cdot d_2)$$

By PAR-INTER-DIST

$$\equiv \sum_{(e_1,d_1)\in D(q),(e_2,d_2)\in D(r)} ([e_1(\alpha)(\beta)] \cap [e_2(\alpha)(\beta)]) \cdot (d_1 \cap d_2)$$

By INTER-FILTER-DIST-LEFT

$$\equiv \sum_{(e_1,d_1)\in D(q),(e_2,d_2)\in D(r)} ([e_1(\alpha)(\beta)] \cap [e_2(\alpha)(\beta)]) \cdot d_1 \cap d_2$$

By INTER-FILTER-DIST-LEFT

$$\equiv \sum_{(e_1,d_1)\in D(q),(e_2,d_2)\in D(r)} ([e_1(\alpha)(\beta)] \cap [e_2(\alpha)(\beta)]) \cdot (d_1 \cap d_2)$$

By INTER-IDEM

$$\equiv \sum_{(e,d)\in\{(e_1,d_1)\cap(e_2,d_2)|(e_1,d_1)\in D(q),(e_2,d_2)\in D(r)\}} [e(\alpha)(\beta)] \cdot d$$

$$\equiv \sum_{(e,d)\in D(p\cap q)} [e(\alpha)(\beta)] \cdot d$$

Case $p = q^*$

$$\begin{aligned}
D'_{\alpha\beta}(q^*) &\equiv \sum_\gamma E_{\alpha,\gamma}(q^*) \cdot D'_{\gamma,\beta}(q) \cdot q^* \\
&\equiv \sum_\gamma E_{\alpha,\gamma}(q^*) \cdot \left(\sum_{(e,d)\in D(q)} [e(\gamma)(\beta)] \cdot d \right) q^* \\
&\equiv \sum_\gamma E_{\alpha,\gamma}(q^*) \cdot \left(\sum_{(e,d)\in D(q)} [e(\gamma)(\beta)] \cdot d \cdot q^* \right) \\
&\equiv \sum_\gamma \sum_{(e,d)\in D(q)} E_{\alpha,\gamma}(q^*) \cdot [e(\gamma)(\beta)] \cdot d \cdot q^* \\
&\equiv \sum_\gamma \sum_{(e,d)\in D(q)} E(q^*)(\alpha)(\gamma) \cdot [e(\gamma)(\beta)] \cdot d \cdot q^* \\
&\equiv \sum_{(e,d)\in D(q)} [(E(q^*) \cdot e)(\alpha)(\beta)] \cdot d \cdot q^* \\
&\equiv \sum_{(e,d)\in D(q)} E(q^*) \cdot [e(\alpha)(\beta)] \cdot d \cdot q^*
\end{aligned}$$

$$\equiv \sum_{(e,d)\in D(q^*)} [e(\alpha)(\beta)] \cdot d$$

Case $p = \overline{q}$

$$\begin{aligned} D'_{\alpha\beta}(\overline{q}) &= \overline{D'_{\alpha\beta}(q)} \\ &\equiv \overline{\sum_{(e,d)\in D(q)} [e(\alpha)(\beta)] \cdot d} \\ &\equiv \prod_{(e,d)\in D(q)} \overline{[e(\alpha)(\beta)] \cdot d} \\ &\equiv \prod_{(e,d)\in D(q) \text{ s.t. } e(\alpha)(\beta)} \overline{d} \\ &\equiv \sum_{(e,d)\in D(\overline{q})} [e(\alpha)(\beta)] \cdot d \end{aligned}$$

□

Lemma 11 (Lemma 2)**.**

$$D_{\alpha,\beta}(p) \equiv \sum_{(e,d)\in D(p)} [e(\alpha)(\beta)] \cdot \beta \cdot d$$

Proof. Follows directly from previous lemma. □

NetKAT$(-,\cap)$ spines To prove that our equivalence checking algorithm terminates, we need to show that there is some finite bound upon the state space of our automata. In the original NetKAT paper, this was shown by demonstrating a finite basis for the state space, called *spines*. We extend their spine construction to NetKAT$(-,\cap)$, and use this extension to show that the set of derivatives of a term is finite, identifying terms up to associativity, commutativity, and idempotency (ACI) of intersection. Unlike NetKAT spines, extended

NetKAT($-, \cap$) extended spines

$$
\begin{aligned}
\mathit{espine}(a) &\triangleq \emptyset \\
\mathit{espine}(f \leftarrow n) &\triangleq \emptyset \\
\mathit{espine}(p + q) &\triangleq \mathit{espine}(p) \cup \mathit{espine}(q) \\
\mathit{espine}(p \cdot q) &\triangleq \{e \cdot q \mid e \in \mathit{espine}(p)\} \cup \mathit{espine}(q) \\
\mathit{espine}(p^*) &\triangleq \{e \cdot p^* \mid e \in \mathit{espine}(p)\} \\
\mathit{espine}(\mathsf{dup}) &\triangleq \{1\} \\
\mathit{espine}(p \cap q) &\triangleq \{p' \cap q' \mid p' \in \mathit{espine}(p) \wedge q' \in \mathit{espine}(q)\} \\
\mathit{espine}(\overline{p}) &\triangleq \cap^*(\{\overline{q} \mid q \in \mathit{espine}(p))
\end{aligned}
$$

spines are not proper subterms of the original term. In fact, the size of the set of extended spines of a term is non-elementary in the size of the original term. Because FDDs are a specific representation of the state space, finiteness of spines implies finiteness of FDD based automata, modulo semantic equivalence of FDDs.

The definition of extended spines (shown in Appendix A.1.2) is not quite as elegant as the original spine definition. NetKAT spines were able to succinctly identify terms up to ACI of + by using a set representation. Because we work with two distinct ACI operations (+ and $\cap$), a single layer of sets does not suffice. Instead, we use sets to capture ACI of + (as in the original spines), and implicitly work with intersections up to ACI. In the actual implementation, this is also implemented using a set representation.

In particular, we work with the intersection closure of a finite set of terms up to ACI. To make this precise, we can uniquely define the intersection closure of a set S ($\cap^*(S)$) by taking the powerset of the set of terms, and then sorting each subset and taking the formal intersection of each. But this level of formality is not necessary to understand the development.

Theorem 19. $\forall (e, d) \in D(p), d \in \mathit{espine}(p)$

Proof. Proof by structural induction upon p.

Case π

$$\begin{aligned} D(\pi) &= \emptyset \\ &= \mathit{espine}(\pi) \end{aligned}$$

Case b

$$\begin{aligned} D(b) &= \emptyset \\ &= \mathit{espine}(b) \end{aligned}$$

Case dup

$$\begin{aligned} D(\mathsf{dup}) &= \{(E(\mathsf{1}), \mathsf{1})\} \\ \{\mathsf{1}\} &= \mathit{espine}(\mathsf{dup}) \end{aligned}$$

Case $p + q$

$$\begin{aligned} D(p+q) &= D(p) \cup D(q) \\ \mathit{espine}(p+q) &= \mathit{espine}(p) \cup \mathit{espine}(q) \end{aligned}$$

The case follows by induction.

Case $p \cap q$

$$\begin{aligned} D(p \cap q) &= \{d_1 \cap d_2 \mid d_1 \in D(p), d_2 \in D(q)\} \\ \mathit{espine}(p \cap q) &= \{e_1 \cap e_2 \mid e_1 \in \mathit{espine}(p), e_2 \in \mathit{espine}(q)\} \end{aligned}$$

The case follows by induction.

Case $p \cdot q$

$$D(p \cdot q) = \{d \cdot q \mid d \in D(p)\} \cup E(p) \cdot D(q)$$

$$espine(p \cdot q) = \{e \cdot q \mid e \in espine(p)\} \cup espine(q)$$

The case follows by induction.

Case p^*

$$D(p^*) = \{d \cdot p \mid d \in D(p)\}$$

$$espine(p^*) = \{e \cdot p \mid e \in espine(p)\}$$

The case follows by induction.

Case $\overline{p}$

$$D(\overline{p}) = \bigcup_{\alpha,\beta} \left\{ (E(\alpha \cdot p_\beta), \bigcap_{(e',d') \in D(p) \wedge e'(\alpha)(\beta)} \overline{d'}) \right\}$$

$$espine(\overline{p}) = \cap^*(\{\overline{q} \mid q \in espine(p))$$

The case follows by induction.

□

A.2 Proofs for Chapter 4

The theorems of this chapter have been formally verified in the Coq theorem prover. We include the proof text in the thesis supplement. All proofs have been completed in Coq version 8.4.

The Coq proofs for this chapter were jointly developed with Arjun Guha. The compiler was based on a system originally built by Arjun Guha and Andrew D. Ferguson.

A.3 Proofs for Chapter 5

The theorems of this chapter have been formally verified in the Coq theorem prover. We include the proof text in the thesis supplement. All proofs have been completed in Coq version 8.4.